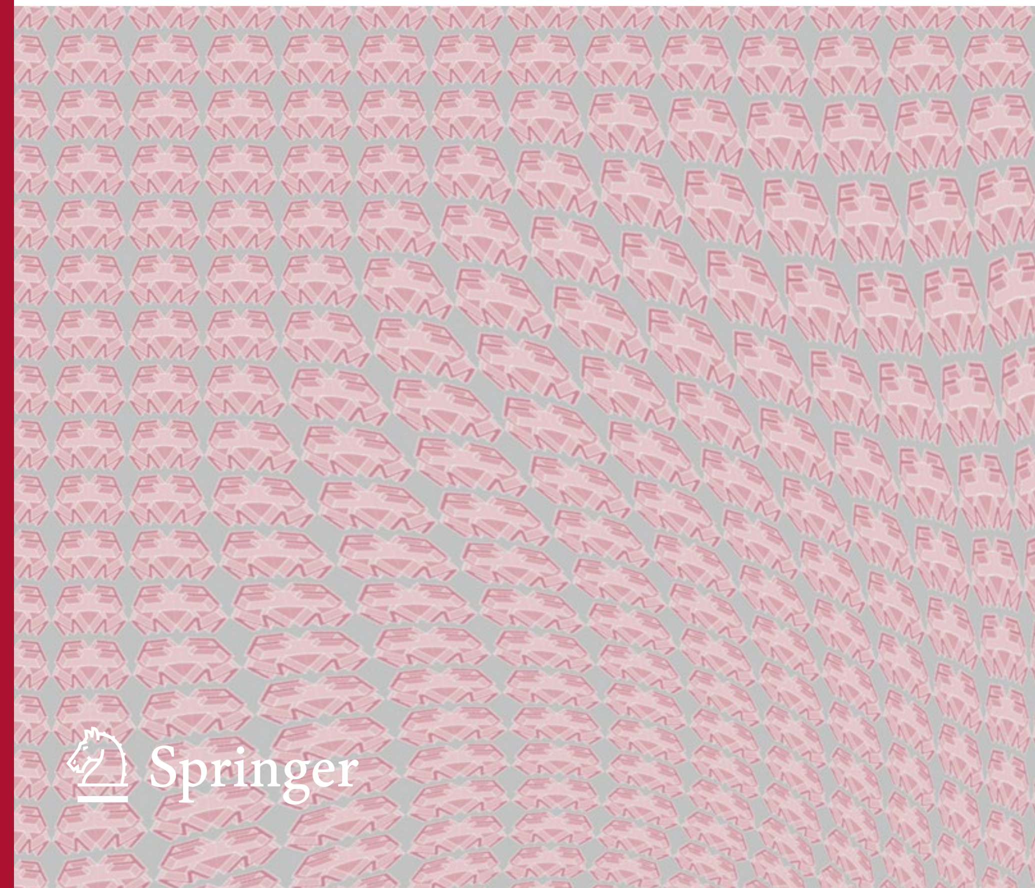
Formal Methods

LNCS 11023

Carlo A. Furia
Kirsten Winter (Eds.)

Integrated Formal Methods

14th International Conference, IFM 2018
Maynooth, Ireland, September 5–7, 2018
Proceedings



Springer

Lecture Notes in Computer Science 11023

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

Editorial Board

David Hutchison, UK
Josef Kittler, UK
Friedemann Mattern, Switzerland
Moni Naor, Israel
Bernhard Steffen, Germany
Doug Tygar, USA
Takeo Kanade, USA
Jon M. Kleinberg, USA
John C. Mitchell, USA
C. Pandu Rangan, India
Demetri Terzopoulos, USA
Gerhard Weikum, Germany

Formal Methods

Subline of Lectures Notes in Computer Science

Subline Series Editors

Ana Cavalcanti, *University of York, UK*
Marie-Claude Gaudel, *Université de Paris-Sud, France*

Subline Advisory Board

Manfred Broy, *TU Munich, Germany*
Annabelle McIver, *Macquarie University, Sydney, NSW, Australia*
Peter Müller, *ETH Zurich, Switzerland*
Erik de Vink, *Eindhoven University of Technology, The Netherlands*
Pamela Zave, *AT&T Laboratories Research, Bedminster, NJ, USA*

More information about this series at http://www.springer.com/series/7408

Carlo A. Furia · Kirsten Winter (Eds.)

Integrated Formal Methods

14th International Conference, IFM 2018
Maynooth, Ireland, September 5–7, 2018
Proceedings

Editors
Carlo A. Furia
Università della Svizzera Italiana
Lugano
Switzerland

Kirsten Winter
University of Queensland
Brisbane, QLD
Australia

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-98937-2 ISBN 978-3-319-98938-9 (eBook)
https://doi.org/10.1007/978-3-319-98938-9

Library of Congress Control Number: 2018950771

LNCS Sublibrary: SL2 – Programming and Software Engineering

© Springer Nature Switzerland AG 2018
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Broadening the adoption and applicability of formal methods hinges on being able to combine different formalisms and different analysis techniques – because different components may be more amenable to different techniques, or simply to express complex properties and to reason about complex systems. The Integrated Formal Methods (iFM) conference series targets research in formal approaches that combine different methods for modeling and analysis. The conference covers a broad spectrum of topics: from language design, to verification and analysis techniques, to supporting tools and their integration into software engineering practice.

This volume contains the proceedings of iFM 2018, which took place in Maynooth during September 5–7, 2018, and was hosted by the National University of Ireland. The Program Committee (PC) received 60 paper submissions. After several weeks of reviewing, followed by a lively online discussion among PC members, we selected 22 contributions (17 regular papers and five short papers) for inclusion in this proceedings volume and presentation at the conference. The combination of topics covered by the selected papers includes both theoretical approaches and practical implementations, demonstrating that the underlying principle of integrating heterogeneous formal methods can buttress rigorous solutions in different domains and at different levels of abstraction.

The scientific program of iFM 2018 was completed by keynote talks given by Cristian Cadar (Imperial College London, UK), Ana Cavalcanti (University of York, UK), and Viktor Vafeiadis (MPI-SWS, Germany), whose content is also documented in this volume. We would like to thank our invited speakers for delivering exciting presentations that served as an inspiration to the iFM community.

We also thank the PC members and the reviewers who helped them for their thorough reviewing work, and for animating a careful discussion of the merits of each submission. Their names are listed on the following pages. The EasyChair system provided indispensable practical support to the reviewing and discussion process.

The local organization in Maynooth ensured a successful and enjoyable conference. We are grateful to all the organizers, and in particular to the general chair, Rosemary Monahan, who took care of all organizational aspects with great resourcefulness and punctuality. Thanks also to Hao Wu for helping with publicizing iFM 2018 and its related events.

Finally, we would like to acknowledge the organizations that sponsored the conference: Maynooth University, Failte Ireland, Science Foundation Ireland, the Embassy of France in Ireland, ACM SIGLOG, and Springer.

July 2018

Carlo A. Furia
Kirsten Winter

Organization

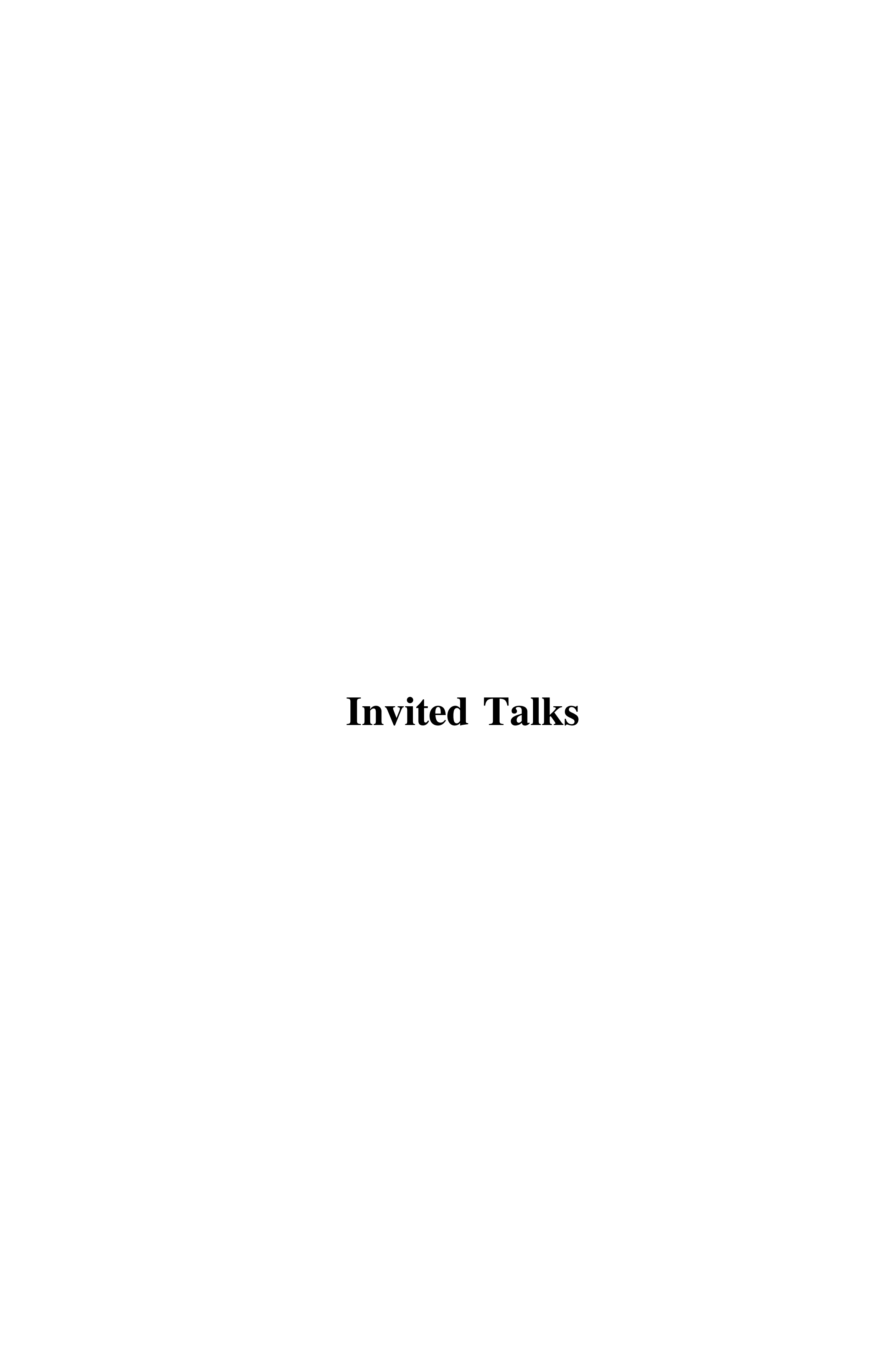
Program Committee

Erika Abraham	Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, Germany
Bernhard K. Aichernig	Technische Universität Graz, Austria
Elvira Albert	Complutense University of Madrid, Spain
Domenico Bianculli	University of Luxembourg, Luxembourg
Eerke Boiten	University of Kent, UK
Einar Broch Johnsen	University of Oslo, Norway
Maria Christakis	Max Planck Institute for Software Systems (MPI-SWS), Germany
David Cok	GrammaTech, USA
Robert Colvin	The University of Queensland, Australia
Ferruccio Damiani	Università di Torino, Italy
Eva Darulova	Max Planck Institute for Software Systems (MPI-SWS), Germany
Frank de Boer	Centrum Wiskunde and Informatica (CWI) Amsterdam, Netherlands
John Derrick	University of Sheffield, UK
Brijesh Dongol	Brunel University, UK
Catherine Dubois	École nationale supérieure d'informatique pour l'industrie et l'entreprise (ENSIIE), France
Carlo A. Furia	Università della Svizzera Italiana (USI), Switzerland
Diego Garbervetsky	University of Buenos Aires, Argentina
Peter Höfner	Data61, The Commonwealth Scientific and Industrial Research Organisation (CSIRO), Australia
Marieke Huisman	University of Twente, The Netherlands
Rajeev Joshi	NASA Laboratory for Reliable Software, Jet Propulsion Laboratory (JPL), USA
Nikolai Kosmatov	CEA Laboratory for Integration of Systems and Technology (LIST), France
Laura Kovács	Vienna University of Technology, Austria
Rustan Leino	Amazon, USA
Larissa Meinicke	The University of Queensland, Australia
Dominique Mery	Université de Lorraine, Lorraine Research Laboratory in Computer Science and Its Applications (LORIA) Nancy, France
Rosemary Monahan	Maynooth University, Ireland
Toby Murray	University Melbourne, Australia
Luigia Petre	Åbo Akademi University, Finland

Ruzica Piskac	Yale University, USA
Chris Poskitt	Singapore University of Technology and Design (SUTD), Singapore
Kostis Sagonas	Uppsala University, Sweden
Gerhard Schellhorn	Universität Augsburg, Germany
Steve Schneider	University of Surrey, UK
Gerardo Schneider	Chalmers - University of Gothenburg, Sweden
Emil Sekerinski	McMaster University, Canada
Martin Steffen	University of Oslo, Norway
Helen Treharne	University of Surrey, UK
Caterina Urban	Eidgenössische Technische Hochschule (ETH) Zürich, Switzerland
Mark Utting	University of the Sunshine Coast, Australia
Heike Wehrheim	University of Paderborn, Germany
Kirsten Winter	The University of Queensland, Australia
Mitsuharu Yamamoto	Chiba University, Japan
Chenyi Zhang	Jinan University, China

Additional Reviewers

Attiogbe, Christian
Bodenmüller, Stefan
Bodeveix, Jean-Paul
Botella, Bernard
Ernst, Gidon
Ghassemi, Fatemeh
Hallahan, William
Isabel, Miguel
Johansen, Christian
Kotelnikov, Evgenii
Lienhardt, Michael
Marcozzi, Michaël
Pfähler, Jörg
Pun, Ka I
Román-Díez, Guillermo
Santolucito, Mark
Schumi, Richard
Sharma, Arnab
Siewe, Francois
Singh, Neeraj
Thüm, Thomas
van Glabbeek, Rob
Whitefield, Jorden
Wüstholz, Valentin



Invited Talks

Dynamic Symbolic Execution for Software Analysis

Cristian Cadar

Imperial College London, UK
c.cadar@imperial.ac.uk

Abstract. Symbolic execution is a program analysis technique that can automatically explore and analyse paths through a program. While symbolic execution was initially introduced in the seventies, it has only received significant attention during the last decade, due to tremendous advances in constraint solving technology and effective blending of symbolic and concrete execution into what is often called dynamic symbolic execution. Dynamic symbolic execution is now a key ingredient in many computer science areas, such as software engineering, computer security, and software systems, to name just a few.

In this talk, I will discuss recent advances and ongoing challenges in the area of dynamic symbolic execution, drawing upon our experience developing several symbolic execution tools for many different problems, such as high-coverage test input generation, bug and security vulnerability detection, patch testing and bounded verification, among many others.

Modelling and Verification for Swarm Robotics

Ana Cavalcanti[1], Alvaro Miyazawa[1], Augusto Sampaio[2], Wei Li[3], Pedro Ribeiro[1] and Jon Timmis[3]

[1] Department of Computer Science, University of York, UK
[2] Centro de Informática, Universidade Federal de Pernambuco, Brazil
[3] Department of Electronic Engineering, University of York, UK

Abstract. RoboChart is a graphical domain-specific language, based on UML, but tailored for the modelling and verification of single robot systems. In this paper, we introduce RoboChart facilities for modelling and verifying heterogeneous collections of interacting robots. We propose a new construct that describes the collection itself, and a new communication construct that allows fine-grained control over the communication patterns of the robots. Using these novel constructs, we apply RoboChart to model a simple yet powerful and widely used algorithm to maintain the aggregation of a swarm. Our constructs can be useful also in the context of other diagrammatic languages, including UML, to describe collections of arbitrary interacting entities.

Program Correctness under Weak Memory Consistency

Viktor Vafeiadis

MPI-SWS, Germany

Abstract. It is fairly common knowledge that shared-memory concurrent programs running on modern multicore processors do not adhere to the interleaving concurrency model, but rather exhibit weakly consistent behaviours, such as store and load buffering. Formally, the semantics of shared-memory concurrent programs is determined by a weak memory model, defined either by the programming language (e.g., in the case of C/C++11 or Java) or by the hardware architecture (e.g., for assembly and legacy C code).

These weak memory models pose two major challenges for software verification. First, many standard proof techniques that were developed for interleaving concurrency, such as the Owicki-Gries method, are unsound in the context of weak memory consistency. Second, it is not even clear how to specify the correctness of concurrent libraries when there is no a globally agreed notion of time and state. To overcome these challenges, we therefore have to develop new techniques for specifying and verifying weakly consistent concurrent programs. The invited talk will present some first steps in this direction.

Contents

Modelling and Verification for Swarm Robotics 1
Ana Cavalcanti, Alvaro Miyazawa, Augusto Sampaio, Wei Li, Pedro Ribeiro, and Jon Timmis

On the Industrial Uptake of Formal Methods in the Railway Domain: A Survey with Stakeholders 20
Davide Basile, Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, Franco Mazzanti, Andrea Piattino, Daniele Trentini, and Alessio Ferrari

Reasoning About JML: Differences Between KeY and OpenJML 30
Jan Boerman, Marieke Huisman, and Sebastiaan Joosten

Design and Verification of Restart-Robust Industrial Control Software. 47
Dimitri Bohlender and Stefan Kowalewski

Efficiently Characterizing the Undefined Requests of a Rule-Based System. 69
Zheng Cheng, Jean-Claude Royer, and Massimo Tisi

Study of Integrating Random and Symbolic Testing for Object-Oriented Software 89
Marko Dimjašević, Falk Howar, Kasper Luckow, and Zvonimir Rakamarić

Making Linearizability Compositional for Partially Ordered Executions 110
Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick

Security Protocol Analysis in Context: Computing Minimal Executions Using SMT and CPSA 130
Daniel J. Dougherty, Joshua D. Guttman, and John D. Ramsdell

A Precise Pictorial Language for Array Invariants 151
Johannes Eriksson, Masoumeh Parsa, and Ralph-Johan Back

Robotics and Integrated Formal Methods: Necessity Meets Opportunity 161
Marie Farrell, Matt Luckcuck, and Michael Fisher

Formal Modelling of Software Defined Networking. 172
Vashti Galpin

Resource-Aware Virtually Timed Ambients 194
Einar Broch Johnsen, Martin Steffen, Johanna Beate Stumpf, and Lars Tveito

Stateful Behavioral Types for Active Objects 214
Eduard Kamburjan and Tzu-Chun Chen

Probabilistic Verification of Timing Constraints in Automotive Systems Using UPPAAL-SMC 236
Eun-Young Kang, Dongrui Mu, and Li Huang

Facilitating the Implementation of Distributed Systems with Heterogeneous Interactions 255
Salwa Kobeissi, Adnan Utayim, Mohamad Jaber, and Yliès Falcone

State-of-the-Art Model Checking for B and Event-B Using ProB and LTSmin 275
Philipp Körner, Michael Leuschel, and Jeroen Meijer

Towards a Formal Notion of Impact Metric for Cyber-Physical Attacks 296
Ruggero Lanotte, Massimo Merro, and Simone Tini

Task Planning with OMT: An Application to Production Logistics 316
Francesco Leofante, Erika Ábrahám, and Armando Tacchella

Branching Temporal Logic of Calls and Returns for Pushdown Systems 326
Huu-Vu Nguyen and Tayssir Touili

Repair and Generation of Formal Models Using Synthesis 346
Joshua Schmidt, Sebastian Krings, and Michael Leuschel

Mode-Aware Concolic Testing for PLC Software: Special Session “Formal Methods for the Design and Analysis of Automated Production Systems” . . . 367
Hendrik Simon and Stefan Kowalewski

Formalisation of SysML/KAOS Goal Assignments with *B System* Component Decompositions 377
Steve Jeffrey Tueno Fotso, Marc Frappier, Régine Laleau, Amel Mammar, and Michael Leuschel

Analysing AWN-Specifications Using mCRL2 (Extended Abstract). 398
Rob van Glabbeek, Peter Höfner, and Djurre van der Wal

Author Index 419

Modelling and Verification for Swarm Robotics

Ana Cavalcanti[1(✉)], Alvaro Miyazawa[1], Augusto Sampaio[2], Wei Li[3], Pedro Ribeiro[1], and Jon Timmis[3]

[1] Department of Computer Science, University of York, York, UK
Ana.Cavalcanti@york.ac.uk
[2] Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil
[3] Department of Electronic Engineering, University of York, York, UK

Abstract. RoboChart is a graphical domain-specific language, based on UML, but tailored for the modelling and verification of single robot systems. In this paper, we introduce RoboChart facilities for modelling and verifying heterogeneous collections of interacting robots. We propose a new construct that describes the collection itself, and a new communication construct that allows fine-grained control over the communication patterns of the robots. Using these novel constructs, we apply RoboChart to model a simple yet powerful and widely used algorithm to maintain the aggregation of a swarm. Our constructs can be useful also in the context of other diagrammatic languages, including UML, to describe collections of arbitrary interacting entities.

1 Introduction

In [15,22], RoboChart, a domain-specific language tailored for robotics, is presented. The core of RoboChart is based on state machines, a modelling construct widely employed in the embedded-software and robotics domains. RoboChart is endowed with a denotational semantics that supports both automatic and semi-automatic verification in the form of model checking and theorem proving.

Unlike general purpose notations, like, for example, UML [12], RoboChart is concise, with well defined syntax and well-formedness conditions that guarantee meaningfulness of models. RoboChart also includes constructs for modelling abstraction (given types, operations definitions via pre and postconditions, and so on), nondeterminism, and time. Most languages of the same nature avoid abstraction and nondeterminism since these features make code generation difficult or impossible. While time is considered in UML MARTE [11] and UML-RT [18], the RoboChart approach based on budgets and deadlines is distinctive.

RoboChart is supported by RoboTool, which provides facilities for graphical modelling, validation, and automatic generation of C++ simulations. RoboTool automatically generates also the formal semantics of RoboChart models. The semantics definition uses the process algebra CSP [20], and RoboTool also provides a direct connection to the FDR model checker for CSP [10].

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 1–19, 2018.
https://doi.org/10.1007/978-3-319-98938-9_1

RoboChart as presented in [15], however, enforces the use of design patterns appropriate for systems composed of a single robot. While such applications are relevant and widespread, collections of robots, that is, swarms, are becoming popular. In robotic swarms, a goal such as pushing a block is achieved by a collection of simple and cheap robots that individually cannot complete the task, but can in cooperation. Their relative low cost allows for defective robots to be easily replaced, making a swarm more robust than single robot applications.

Here, we extend RoboChart to support modelling and verification of collections of interacting robots. We focus on the nature of the robots, and how they communicate with each other. With these, we can specify the behaviour of a swarm as the result of the interaction of a (n unspecified) number of robotic systems. We can describe abstractly heterogeneous collections of interacting robots through the use of underspecified constants, different robot specifications, and communication patterns. We introduce a new inter-robot communication mechanism for fine-grained control over interactions, supporting both identification of the source of interactions and restriction of the possible targets.

Although our focus is on capturing precisely descriptions of swarm applications from the robotics literature, our modelling constructs can be useful to model arbitrary heterogeneous distributed systems. As far as we know, our constructs are entirely novel. In UML or SysML [17], for instance, a variant of UML for systems modelling, the definition of a collection and their connections requires two diagrams, and fixes the number of components in the collection.

We note, however, that as a design language, RoboChart does not cover the explicit specification of global properties of the swarm, such as aggregation. A property language for RoboChart is part of our agenda for future work.

Section 2 briefly introduces the RoboChart notation by means of a simple example of an aggregation algorithm running on a single robot. Section 3 describes the extensions necessary to accommodate the new features, and Sect. 4 describes their semantics. Section 5 reviews the tool support available for RoboChart and its extensions. Section 6 discusses related work. Finally, Sect. 7 concludes and discusses further opportunities for work.

2 RoboChart and Its Semantics

Here, to illustrate the RoboChart notation, we present in Sect. 2.1 a model of the alpha algorithm [4][1], whose goal is to maintain a collection of robots in an aggregate. This algorithm estimates the number of neighbours of a robot, and uses that to decide whether to maintain its direction or turn around. The idea is that the robot recognises when it has moved away from the aggregate by counting the number of neighbours. Later, we show how our support for collections allows for a much simpler and clearer model. In Sect. 2.2, we briefly introduce CSP. Finally, Sect. 2.3 gives an overview of the semantics of RoboChart.

[1] www.cs.york.ac.uk/circus/RoboCalc/case-studies/.

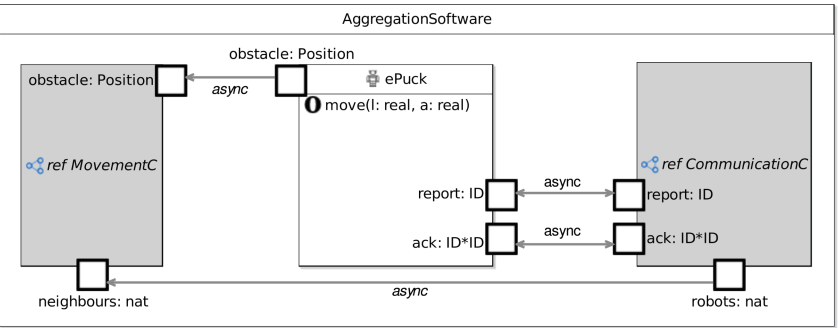


Fig. 1. A robot implementing an aggregation algorithm.

2.1 The Notation

A RoboChart model describes two main aspects of an application: structure and behaviours. The root element of a model is called a module, and provides an overall view of the system. The module for our example is in Fig. 1. A module specifies two aspects of the applications: (1) assumptions about the platform, and (2) available behaviours. The assumptions are modelled via a robotic-platform block, illustrated in Fig. 1 by the block ePuck, which describes the variables, operations, and events that must be available for the application to be feasible.

In this example, the platform ePuck abstracts a piece of hardware containing a number of sensors and actuators. The operation move that takes two parameters l and a, both of type real, models the actuator responsible for moving the robot forward and turning. The event obstacle represents an obstacle sensor and communicates the position of the obstacle. The events report and ack represent sensors and actuators responsible for inter-robot communication.

The event obstacle carries a value of type Position that is defined by an enumeration containing the values left and right. The event report carries a value of type ID representing the source of the communication, and ack carries a pair of ID values representing the source and target of the communication. ID is an abstract type about which nothing is assumed except for its non-emptiness.

The behaviours of a RoboChart module are specified by one or more controllers; they run in parallel and interact with each other and with the robotic platform. The possible interactions are indicated by connections, which, at the level of modules, are either synchronous or asynchronous.

In our example, the aggregation behaviour is decomposed into two controllers: MovementC and CommunicationC. The first describes how the robot moves based on the number of neighbours; the second uses inter-robot communications to estimate that number. There are four interaction points: the occurrence of the event obstacle represents an interaction between the robotic platform and MovementC to communicate the position of the obstacle with respect to the robot; report and ack are used to interact with other robots, intermediated by the robotic platform, and communicating the identity of the robot; finally robots

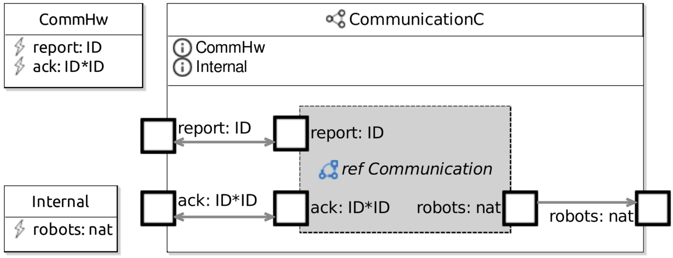


Fig. 2. Communication controller of the aggregation algorithm.

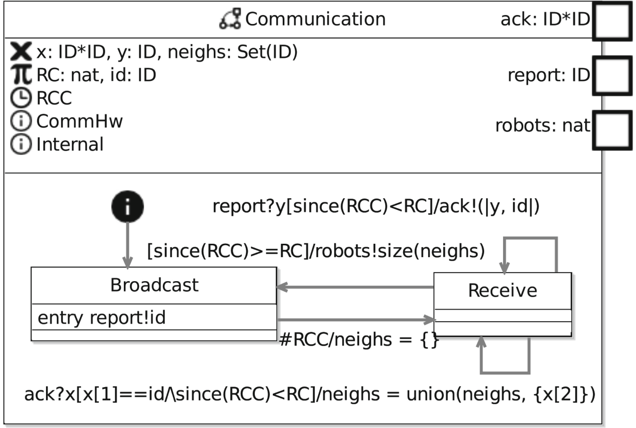


Fig. 3. Communication state machine of the aggregation algorithm.

in CommunicationC is used for interaction with MovementC through neighbours. (Connected events do not need to have the same name, just the same type.)

Controllers are defined by one or more state machines interacting via synchronous connections. Figure 2 shows CommunicationC. It declares two interfaces CommHw and Internal, also shown in Fig. 2. They define the events report, ack and robots, connected to identically named events in the referenced state machine Communication, whose definition is shown in Fig. 3. Just like the connection between CommunicationC and the platform, the connections for report and ack are bidirectional. The connection for robots provides an output, matching the connection between CommunicationC and MovementC.

Communication models a cyclic behaviour, where each cycle takes RC time units. At each cycle, the machine indicates its presence via report, and then monitors for responses from other robots through ack. At the same time, reports from other robots are acknowledged. This allows estimating how many robots are within the communication range. The size of the set of neighbours is sent through robots and propagated to MovementC to decide how to move.

Similarly to the controller CommunicationC, the machine Communication declares the two interfaces CommHw and Internal to define the events it uses. Additionally, it declares two constants: id of type ID, representing the identifier of the robot and RC with type nat of natural numbers. There are also three variables: x of type ID*ID (pair of values of type ID) records values received through ack, y of type ID records values received through report, and neighs of type Set(ID) (sets of values of type ID) records the identifiers of the robots that respond to the event report. Finally, Communication declares a clock RCC.

Communication initially enters the state Broadcast executing its entry action, which sends id through report. After that, the transition to the state Receive is taken, since it has no guard or event. This resets the clock (#RCC) and assigns the empty set ({}) to neighs. From Receive, there are three possible transitions. If the clock runs out (since(RCC)>=RC), the size of neighs is communicated via robots, and Broadcast is entered. If, before the clock runs out (since(RCC)<RC), the event report occurs, the pair (|y,id|) formed input y taken and id is sent through ack, and the machine stays in Receive. Finally, if an event ack, where the first element x[1] of its tuple input x is id, happens before the clock runs out, the second element of x is added to neighs, and the machine stays in Receive.

The semantics of RoboChart captures this behaviour and that of the whole module as a CSP process. We next give a brief overview of CSP.

2.2 CSP

Systems and components are modelled in CSP via processes. They are all regarded as black boxes and defined by the patterns of their interactions the environment. Interaction is via atomic and instantaneous events.

Accordingly, the semantics of a RoboChart model is defined by a process that captures the behaviour of its module. Each component, controller, and state machine is defined by a process as well. The events of the module process match the interface of the robotic system characterised by the robotic platform. They correspond to accesses to variables, to calls to operations of the platform, and to communications using the RoboChart events of the platform.

We explain the CSP notation as we use it; Table 1 gives a summary. A core operator is prefixing $c \rightarrow P$; it describes a process that engages in a communication (event) c and then behaves as the process P. The event may be an input $c?x$ that records the value input via a channel c in a variable x, an output $c!e$ of the value of an expression e, or a simple synchronisation on c.

The parallel operators are also very important. $P \mathbin{[\![} cs \mathbin{]\!]} Q$ defines the parallel execution of P and Q, synchronising on the events in the set cs. Communications internal to a component or system can be hidden using the operator $P \setminus cs$, which hides the events in the set cs in the execution of P.

A dialect of CSP, called tock-CSP, uses a special event *tock* to mark the passage of time. We use tock-CSP to capture the semantics of the timed constructs of RoboChart: clocks, budgets, and deadlines.

Most importantly for our agenda of work, CSP has a relational predicative semantics defined using the Unifying Theories of Programming (UTP) [13].

Table 1. Summary of CSP operators

Symbol	Name	Description
$Skip$	Skip	Terminate immediately without any side effects
$P \lbrack\!\lbrack\, cs \,\rbrack\!\rbrack\, Q$	Parallel composition	Run P and Q in parallel synchronising on events in cs
$P \mid\mid\mid Q$	Interleaving	Run P and Q in parallel without synchronisation
$\{\mid e \mid\}$	Channel set	Set of all possible events associated with channel e
$c \rightarrow P$	Prefix	Synchronise on channel c and then behave like P
$P \setminus cs$	Hiding	Run P with events in cs hidden
$P[\![c \leftarrow d]\!]$	Renaming	Rename the occurrences of event c to d in P
$\mid\mid\mid\, i : I \bullet P(i)$	Replicated interleave	Run $P(i)$ in parallel for all i in I without synchronisation

Support for this semantics in Isabelle/HOL [8] means that our CSP semantics for RoboChart is a front-end for a UTP theory. With that, we can carry out verification using theorem proving. For large models, and for swarm models in particular, this is crucial to ensure scalability.

2.3 Semantics

The module process is defined by the parallel composition of the processes that model its controllers, where events are renamed and the synchronisation set is constructed so that the controllers interact according to the module connections. Asynchronous connections between controllers are modelled by single-cell buffers. For example, the semantics of the module in Fig. 1 is as follows.

$$
\begin{array}{l}
AggregationRobot = \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
MovementC[MovementC_obstacle.out \leftarrow ePuck_obstacle.in] \\
\quad \mid\mid\mid \\
CommunicationC[\; \begin{array}{l}
CommunicationC_report.out \leftarrow ePuck_report.in, \\
CommunicationC_report.in \leftarrow ePuck_report.out, \\
CommunicationC_ack.out \leftarrow ePuck_ack.in] \\
CommunicationC_ack.in \leftarrow ePuck_ack.out]
\end{array}
\end{array} \right) \\
\quad [\![\{\mid CommunicationC_robots, MovementC_neighbours \mid\}]\!] \\
Neighbours_Buffer(\langle\rangle)
\end{array} \right) \\
\quad \setminus \{\mid CommunicationC_robots, MovementC_neighbours \mid\}
\end{array}
$$

AggregationRobot composes in parallel the controller processes *MovementC* and *CommunicationC*, with their channels that represent events connected directly to the platform, that is, *CommunicationC_report* and *CommunicationC_ack*, renamed to the platform channels *ePuck_report* and *ePuck_ack*. For each event, we have a channel that takes tokens *in* and *out* that identify the direction of communication: *in*put or *out*put. The renamings identify the events to establish the connections between the controllers and the platform. The inputs of the platform are identified with the outputs of the controllers and vice-versa. For an unidirectional connection, like that for obstacle, just one renaming is needed.

In the above example, the controller processes do not communicate because they do not interact synchronously. So, the parallelism is an interleaving ($|||$).

The parallel composition of the controller processes is composed in parallel with a buffer process $Neighbours_Buffer(\langle\rangle)$ that records RoboChart events input through a channel *CommunicationC_robots* modelling the RoboChart event robots, and sends them through another channel *MovementC_neighbours* modelling neighbours. Communications using these channels are hidden as the RoboChart events they represent are internal to the module (see Fig. 1). The set $\{\!| c |\!\}$ contains all events that represent communications over the channel c.

The semantics of controllers is similarly defined as the parallel composition of the processes that model its state machines. Since all connections between state machines are synchronous, there is no need for buffers. Channels corresponding to events of the state machine connected to the controller are renamed to channels of the controller. In our example, since the controller CommunicationC contains only one state machine, only the renamings take place.

$$
\begin{array}{l}
CommunicationC = \\
\quad Communication[\; Communication_robots \leftarrow CommunicationC_robots \\
\qquad\qquad\qquad\quad\;\; Communication_report \leftarrow CommunicationC_report \\
\qquad\qquad\qquad\quad\;\; Communication_ack \leftarrow CommunicationC_ack]
\end{array}
$$

The renamings establish the connections between the events of the controller CommunicationC and those of the state machine Communication.

The processes for state machines are defined in a compositional way in terms of processes for states and transitions. They capture the control flow defined by the machine in terms of CSP events that represent accesses and updates to variables required or provided by the machine, calls to operations, and occurrence of RoboChart events. Compositionality is important for verification and can be achieved because we rule out constructs like inter-level transitions, for example. The complete semantics of RoboChart is described and formalised in [22].

Next, we discuss the metamodel of the new constructs to deal with collections, and present an updated version of our example that uses these constructs.

3 Collections in RoboChart: Overview and Metamodel

Our example models the controller of a single robot, but the application itself is a swarm, where multiple robots communicate to estimate their numbers of neigh-

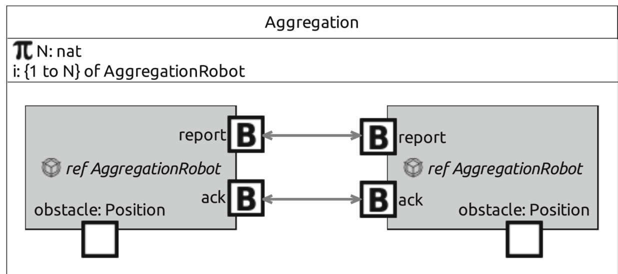


Fig. 4. Collection for the aggregation system.

bours. To capture this behaviour, in the previous section we explicitly model the source and destination of the event ack and the source of the event report.

To model the swarm as a whole and simplify the specification of the communications, we include two new features in RoboChart. We have a construct to specify (heterogeneous) collections, and communications that support extraction and restriction of attributes, namely, source and target robots.

Here, we describe our reworked example (Sect. 3.1), and discuss the collection (Sect. 3.2) and the extended communication (Sect. 3.3) mechanisms.

3.1 Extended Example

In the previous section, the model of our example focused on an individual robot. Here, we model how multiple robots interact with each other, and show how the original model can be reworked to use our extended communication mechanism. We model a swarm of N robots interacting through broadcast events.

Figure 4 shows a collection diagram for our example. It defines the collection as N instances of the module AggregationRobot indexed by values i ranging from 1 to N, where N is an uninitialised constant of type nat. The index specified in the collection is used to identify its individual robots.

Within the bottom compartment of the collection diagram, there are two placeholders for AggregationRobot, with their events report and ack connected. The placeholders stand for any two distinct instances of AggregationRobot. They show how any two robots modelled by AggregationRobot interact.

While communication within a robot (between controllers and state machines) is one-to-one, communication between robots in a collection is a broadcast: all instances of a module with a broadcast event can potentially receive messages from all other instances that are connected to that event as defined in the collection. These events are indicated by the letter B inside the event box.

This system of idealised broadcast connections may seem too restrictive, but, in conjunction with the communication mechanisms discussed in Sect. 3.3, it can be used to model more constrained forms of interaction. We can, for example, model that communication is only sent to a subset of other robots, or even to a particular robot. These can capture the fact that there may be robots out of

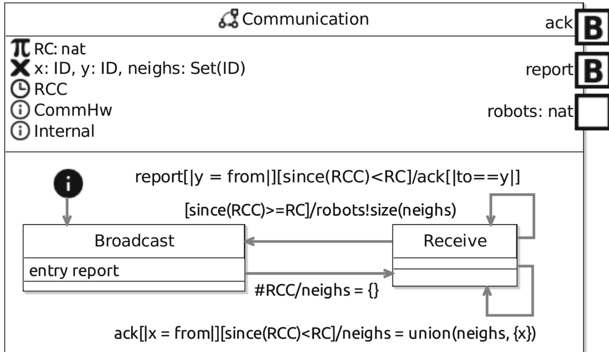


Fig. 5. Broadcast communication in the Communication state machine.

range, or that there is a communication device in the platform with a protocol that identifies when the communications are directed to its robot.

A broadcast event has implicit from and to attributes of a generic type ID to identify the source and target of the communication. The type ID is that of robot identifiers; it is instantiated in a collection diagram via the definition of instances of modules (robots). The diagram in Fig. 4, for example, defines instances with identifiers i in the range 1 to N. In doing so, it instantiates the type ID for AggregationRobot to the set of natural numbers between 1 and N.

Using the collection communication mechanism, we can rewrite the machine in Fig. 3 to eliminate the parameters of the report and ack events, using the attributes of the broadcast events instead. The updated version is in Fig. 5.

In this version, the transition triggered by a report event uses an assignment to record its attribute from in the variable y. It is used in the action that sends the event ack to restrict its attribute to in a predicate that equates to to y. We use a predicate, not an assignment to restrict to, so that, in general, several robots can be targetted. We record the source of the report to direct the acknowledgement to the right robot, since the scope of from and to is the broadcast event.

Similarly, upon receipt of the event ack the source is stored in x. It is that identifier x that is recorded in the set neighs. We note, in particular, that the condition x[1]==id in the guard of the corresponding transition in Fig. 3, becomes unnecessary, since ack is accepted only when the target is the robot id.

In what follows, we describe the metamodel and well-formedness conditions for the collection diagrams and broadcast communications.

3.2 Collections

A collection diagram, illustrated in Fig. 4, describes exactly which types of robots (RoboChart Modules) form the collection and how many instances of each type exist. Furthermore, it specifies how different instances or types of robots can communicate using connections of broadcast events.

In the metamodel of RoboChart, a collection diagram is modelled by a construct RCCollection shown in Fig. 6. It can contain four components. A variableList declares the constants used to define the number of instances of each type of robot. In the metamodel, they are identified as variables, but a well-formedness condition ensures that they are constants. The value of a constant may or may not be defined, like in our example in Fig. 4.

Collections can also have Instantiations consisting of an index with a range, of the Module being instantiated, and of parameters that initialise unspecified constants in the module, if any. For example, we could define the number alpha of neighbours used as a threshold to determine whether a robot should turn as a constant in AggregationRobot. In this case, we may define a value for alpha in the collection, perhaps in terms of the number N of robots in the swarm.

A well-formedness condition ensures that the range is defined by an Expression that denotes a finite set. We note that a collection may have just one instance of a particular Module, so the range may be a singleton. This may be, for example, a model for a collection that contains one robot controlling others.

Another well-formedness condition ensures that the parameters defined via InstantiationParameters are values for constants of the Module instantiated. There may be have several instantiations with different values for these constants.

Placeholders correspond to generic instances of modules. They are references to a Module, that is, an element of ModuleRef. A well-formedness restriction ensures that these are modules that occur in one of the instantiations. An extra attribute IDInst of Module records an instantiation, if any, of the generic type ID of identifiers for instances of the robots defined by the Module.

Since ID is a generic type, there are no operations beyond equality and inequality that can be used to manipulate its elements. If more operations are needed, ID needs to be instantiated. For example, if the models in a component (module, controller, or machine) use arithmetic operations on identifiers, the type ID needs to be instantiated to a numeric type. A well-formedness condition ensures that the same instantiation is used in the collections that use the module. Different instantiations cannot be used in the same context.

Finally, we can include Connections between the placeholders establishing the possible interactions between instances. Connections are between ConnectionNodes, one of which is a ModuleRef. A Connection can be bidirectional, and in a collection, a well-formedness guarantees that it is asynchronous.

If there are several instantiations for the same Module, giving different values for its constants, there may be more than two placeholders for such a Module. In general, for each instantiation, there may be up to one or two placeholders for its Module, depending on whether the range is a singleton or a larger set. An empty range has a well defined semantics that can be used to explore behaviour in the presence of missing robots, perhaps due to failure. Normally, however, we expect that such a range is left unspecified, so that the collection may have instances of the robot defined by the Module in some scenarios.

Events have a type that define the values that can be communicated. In addition, broadcast events model the form of communication used in swarms.

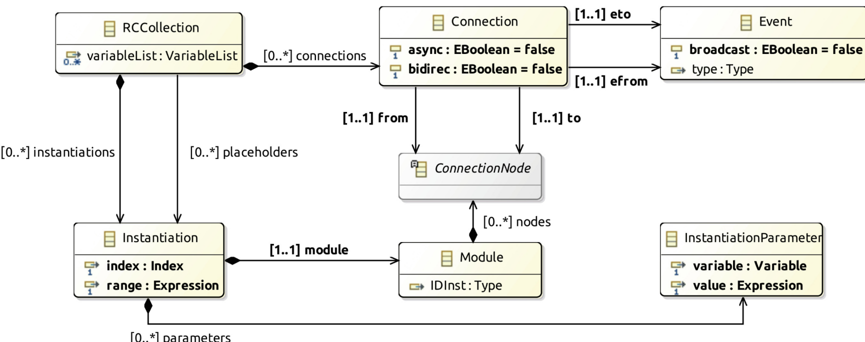


Fig. 6. Metamodel of collections.

Well-formedness requires that all connected events in a collection are broadcast events, and broadcast events are only connected to other broadcast events.

The complete metamodel of RoboChart is available at [22], where elements omitted in Fig. 6, like Type or Expression, are defined.

3.3 Communications

The collection construct provides a general view of the potential communication patterns between robots. In particular, it clearly specifies which communications cannot happen. For example, in Fig. 4, it is specified that there are no interactions using obstacle. Nevertheless, the actual communication pattern cannot be specified at this level. For instance, in our example, while report is used to communicate with all other robots in the swarm, ack is used in a more restricted fashion to communicate only with the robot from which a report has been received.

Although a communication device available in a platform may not be able to enforce a restricted protocol like this, it is simple to program functionality to provide directed communication. The possibility to specify interactions that follow a particular protocol allows us to construct more abstract models, where the programming of any particular protocol is assumed to be in place.

The restricted interactions need to be specified at the level of the communication definitions using intrinsic information about their source and target. Triggers, which are used in transitions and actions to define communications, include extra source and target attributes _from and _predicate. They allow both identification and restriction of the participants.

The attribute _from records the value of a predefined variable from: the identifier of the source of the communication. The _predicate attribute defines a restriction on the target of the communication identified by a predefined variable to. A well-formedness condition ensures that to is free in _predicate. The variables from and to have type ID and are local to a Trigger whose event is a broadcast.

These extensions to Trigger to consider the predefined variables from and to affects the usage of events in triggers in transitions and actions. In an input c?x[|v = from|], the occurrence of the communication c?x is accepted, and the identifier of the source of the communication is recorded in the variable v. An output c!e[|to in S|], for example, sends the value of e via c to each instance specified by the collection diagram whose identifier belongs to the set S.

In the next section, we define the semantics of our collection constructs.

4 Collections in RoboChart: Semantics

The semantics of RoboChart is given by a function $[\![_]\!]_{\mathcal{M}}$ from RoboChart modules to CSP processes. This function is defined compositionally over the metamodel of RoboChart, in terms of other functions that calculate the semantics of the controllers and connections that form the module.

The timed semantics is also defined by a function from RoboChart modules, but its range is the set of tock-CSP processes. The definition of this function reuses much of the original (untimed) semantics. In particular the description of the collection semantics we provide here is valid in the context of both semantics. We note that a tock-CSP process is itself a CSP process.

We define a new semantic function from RCCollection to CSP processes, and modify three functions of the original semantics. The function $[\![_]\!]_{\mathcal{M}}$ is modified just to add a parameter *id* to the module process to record the module identifier. It is used by the processes defined by the functions $[\![_]\!]_{Trigger}$ and $[\![_]\!]_{Statement}$, which specify the semantics of triggers in transitions and in actions.

The definitions of $[\![_]\!]_{Trigger}$ and $[\![_]\!]_{Statement}$ deal with the extra properties _from and _predicate of triggers. In addition, events corresponding to accesses to variables of the platform, to calls to its operations, and to simple platform events, that is, that are not for broadcast, get *id* as an extra parameter. This is so that the individual interactions of the platform with the controller and the environment can be distinguished. Before we discuss all the affected semantics functions, we present the CSP process *Aggregation* below that defines our example collection to illustrate the overall idea of the semantics.

$$Aggregation = \left(\begin{array}{l} ||| \; i : 1..N \bullet AggregationRobot(i) \\ \quad [|\{|report.in, report.out, ack.in, ack.out|\}|] \\ ||| \; (i,j) : 1..N \times (1..N \setminus \{i\}) \bullet \\ \quad \left(Buffer(\langle\rangle, report, i, report, j) \; ||| \; Buffer(\langle\rangle, ack, i, ack, j) \right) \end{array}\right)$$

The robots do not communicate directly. Using, for example, infrared or radio devices, they communicate asynchronously. So, formally, they communicate via buffers. Accordingly, in *Aggregation*, we combine the instances with identifiers 1 to N in interleaving. Instantiation defines the parameter *id* of the module.

There is a buffer for each direction of each connection between each pair of robots. So, for each pair (i, j) of robots i and j, we have a buffer to connect their report and their ack events. We note that i and j are different robots, since j is taken from the set $(1 \,..\, N \backslash \{i\})$, which excludes i. $Buffer(elems, e_1, id_1, e_2, id_2)$

Rule 1. Semantics of Collections $[\![\mathsf{c : RCCollection}]\!]_{\mathcal{C}ol} : \mathsf{CSPProcess} =$

$$
\begin{array}{l}
|||\ \mathsf{inst : c.instantiations} \bullet |||\ i : \mathsf{inst.range} \bullet [\![\mathsf{inst.module}]\!]_{\mathcal{M}}(i)\\
\quad [| \{| e_1, e_2 \mid (e_1, e_2) \leftarrow \mathsf{connectedEvents(c)} |\} |]\\
\left(\begin{array}{l}
\left(\begin{array}{l}
|||\ \mathsf{conn : c.connections} \bullet |||\ (i, j) : \mathsf{inds(conn, c)} \bullet\\
\quad \mathit{Buffer}(\langle\rangle, \mathsf{eventId(conn.efrom)}, i, \mathsf{eventId(conn.eto)}, j)
\end{array}\right)\\
|||\\
\left(\begin{array}{l}
|||\ \mathsf{conn : c.connections} \mid \mathsf{conn.bidirec} \bullet |||\ (i, j) : \mathsf{inds(conn, c)} \bullet\\
\quad \mathit{Buffer}(\langle\rangle, \mathsf{eventId(conn.eto)}, j, \mathsf{eventId(conn.efrom)}, i)
\end{array}\right)
\end{array}\right)
\end{array}
$$

where

$$
\begin{array}{l}
\mathsf{connectedEvents(c : Collection)} : \mathbb{P}(\mathsf{Event} \times \mathsf{Event}) =\\
\quad \{\mathsf{conn : c.connections} \bullet (\mathsf{eventId(conn.efrom)}, \mathsf{eventId(conn.eto)})\}\\
\quad \cup\\
\quad \{\mathsf{conn : c.connections} \mid \mathsf{conn.bidirec} \bullet (\mathsf{eventId(conn.eto)}, \mathsf{eventId(conn.efrom)})\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{inds(conn : Connection, c : Collection)} : \mathbb{P}(\mathsf{ID} \times \mathsf{ID}) =\\
\quad \mathsf{if\ conn.from.ref = conn.to.ref\ then}\\
\qquad \mathsf{range(conn.to, c)} \times \mathsf{range(conn.from, c)} \setminus \{i : \mathsf{range(conn.to, c)} \bullet (i, i)\}\\
\quad \mathsf{else}\\
\qquad \mathsf{range(conn.to, c)} \times \mathsf{range(conn.from, c)}
\end{array}
$$

$$
\mathsf{range(m : Module, c : Collection)} : \mathbb{P}\,\mathsf{ID} = (\iota\, \mathsf{i : c.instantiations} \mid \mathsf{i.module = m}).\mathsf{range}
$$

defines a buffer containing the elements in the sequence *elems*, taking inputs from robot id_1 via the events $e_1.in$ and producing outputs to robot id_2 via events $e_2.out$. The buffers are initially empty: their sequence of elements is $\langle\rangle$.

The buffers do not interact with each other, so they are also combined in interleaving. Their inputs and outputs are connected to the events of the module processes. So, the module processes and the buffers are composed in parallel synchronising on the events *report.in*, *report.out*, *ack.in*, and *ack.out* corresponding to the ends of the bidirectional connections in Fig. 4.

Rule 1 defines the semantic function $[\![_]\!]_{\mathcal{C}ol}$ that specifies the CSP processes for collections. We use a simple meta-notation based on CSP itself; its terms are underlined. CSP terms are in the usual mathematical font.

The function $[\![_]\!]_{\mathcal{C}ol}$ takes an element c of the type RCCollection defined in the metamodel and returns an element of CSPProcess, a CSP process, defined by the parallel composition of two interleavings as illustrated above. For each instantiation inst of c, that is, in the set c.instantiations, we have a replicated interleave, which is itself combined in interleaving with the interleaving for the other instantiations. For each index i in the range inst.range of the instantiation, we have a process $[\![\mathsf{inst.module}]\!]_{\mathcal{M}}(i)$ combined in interleaving, where $[\![\mathsf{inst.module}]\!]_{\mathcal{M}}$ defines the semantics of the referred module inst.module. This semantic function is as defined in [22] and previously illustrated in Sect. 2, but now the process it defines includes the extra parameter *id* as explained above.

The second interleaving is of buffer processes. There are two groups of such processes: the first models unidirectional communication, and the second complements the first with buffers for the reverse direction of communication for bidirectional connections. In each case, for each connection conn of c (from the set c.connections), we have a replicated interleaving of processes modelling buffers. We have a buffer for each pair of instances of each module connected by conn.

The replicated interleavings are indexed by pairs (i, j) in the set inds(conn, c) containing identifiers of the instances of the modules that can communicate through the connection conn. The definition of inds(conn, c) presented in Rule 1 takes into account the module of the two connected instances. If they are different (conn.from.ref = conn.to.refmodules is false), the pairs in inds(conn, c) are those in the cartesian product of the indices of the source and target modules. Otherwise, we need to discard the identity pairs (i, i), since there is no connection associating an instance of a module to itself.

The set of indices is specified using the function range. It takes a module m and a collection c in which it is instantiated, determines the unique (ι) instantiation whose module is m, and returns the associated range.

The second group of interleavings corresponding to the bidirectional connections is similar, except that the parameters of the process *Buffer* are reversed.

The two top-level interleavings are composed in parallel synchronising on the events corresponding to the source and target of the connections. This is calculated by the function connectedEvents, which takes a collection c and returns a set of pairs of events. It is presented in Rule 1. For each connection in c, we have a pair formed of the channels that model the source event (efrom) and the target event (eto) of the connection. For bidirectional connections (conn.bidirec), we also include the reversed pair in connectedEvents(c).

The semantics of triggers uses the *in* and *out* components of the broadcast channels as well as the source and target identifiers to coordinate the exchange of events between the module instances. To illustrate, we present the semantics of two communications in our example in Fig. 5.

The semantics of report[|y = from|] is given by the CSP process below, which is used to define the semantics of the state machine, itself used as a component in the module process and, therefore, in the collection. Here, *id* is the identifier of the module defined, as explained above, as a parameter of its process.

$$report.out?from!id \rightarrow set_y!from \rightarrow Skip$$

If the buffer process for the connection to the report event of a robot *id* contains a value, it can synchronise with this process. It accepts the source *from* of the connection as input, and assigns its value to the variable y using a communication on a channel *set_y*, and terminates (*Skip*). Variables in state machines are held in memories represented by processes with *set* and *get* channels.

The semantics of $\mathsf{ack}[|\mathsf{to} == \mathsf{y}|]$ is given by the CSP process below.

$$|||\ t : \{to \mid to \leftarrow ID, to == y\} \bullet ack.in!id?t \rightarrow Skip$$

While the trigger process for the previous example only synchronises with one of the buffers, the above process synchronises with all buffers whose target identifiers t satisfy the predicate $t == y$. As previously indicated, the CSP trigger processes illustrated above interact with the buffers in *Aggregation*.

The definition of the semantics of triggers requires a simple change to what is presented in [22] to specify enriched processes like those shown above, which can record identifier information, and synchronise with various buffers.

Validation of the semantics just described is provided by its mechanisation RoboTool, which is presented in the next section.

5 Tool Support

Tool support for RoboChart is implemented in RoboTool[2]. It provides a graphical editor for RoboChart models, a parser for the textual elements of the graphical notations (expressions, statements, transition labels, and so on), validators that check well-formedness conditions, and code generators. These tools are implemented as Eclipse[3] plugins using the Xtext[4] and Sirius[5] frameworks.

The RoboTool graphical editor is shown in Fig. 7. It has been enriched with facilities for the creation of collections, and parsing of collection declarations and of the new triggers. The validator has been extended to check for well-formedness conditions for collections described in Sect. 2.

RoboTool provides three code generators: for the untimed semantics, the timed semantics, and a C++ simulation. The CSP files that are generated can be opened from RoboTool directly into FDR to verify properties of the models. A number of assertions of classical properties are also automatically generated for checking. The code generator for the untimed semantics has been updated to implement the semantics of collections of the previous section, and the update of the remaining code generators is part of our future work.

Using RoboTool and FDR, we have checked deadlock freedom for our example collection. For the empty collection, with $N = 0$, we have a deadlock: if there is no working robot, there is no observable behaviour. Similarly, for $N = 1$, we have a deadlock, because there is no other robot to accept a *report*. In the analyses for larger values of N, there is no deadlock, as expected[6]. Although we have been able to analyse this example, analysis of swarm applications requires theorem proving and our approach is well suited for that. Ongoing work allows automated proof of deadlock freedom using Isabelle/UTP.

[2] https://www.cs.york.ac.uk/circus/RoboCalc/robotool/.
[3] www.eclipse.org.
[4] www.eclipse.org/Xtext.
[5] www.eclipse.org/sirius/.
[6] www.cs.york.ac.uk/circus/RoboCalc/case_studies/.

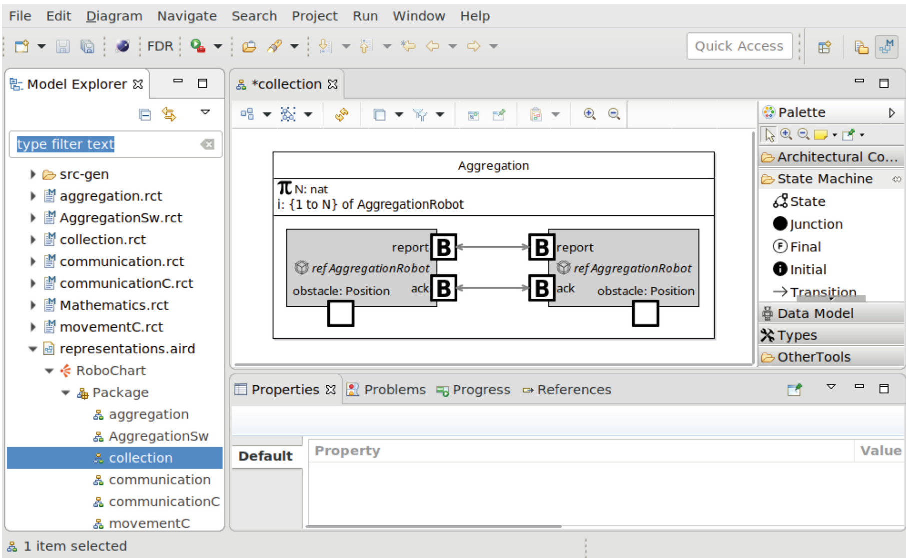


Fig. 7. Graphical editor for RoboChart.

6 Related Work

UML and its various extensions are widely used for modelling in a number of different domains. Due to its generality, UML is an option for modelling robots and swarms. However, while a number of formalisations have been proposed for UML using techniques such as graph transformations [14], CSP [2,19], as well as tailored semantic domains [1], these formalisations only cover subsets of UML.

RoboChart, on the other hand, is a small language, with a well defined process algebraic semantics suitable for verification using model checking and theorem proving. In addition, it caters for timed properties and now has specialised notation for collections of robots. Nordmann *et al.* [16] indicates that domain-specific languages (DSL) for robotics are growing in popularity, further motivating our choice of a small DSL over a more general notation such as UML.

Indeed, several robotic modelling notations have been proposed, but they mostly aim at code generation for execution or simulation. In RoboChart, we also focus on a formal semantics for verification and generation of sound simulations.

RobotML [3] is a UML-based notation for robotics that supports automatic code generation, but support for formal verification is not yet available. Schlegel *et al.* [21] propose the use of a UML-based framework for engineering robotic systems, but formal verification is also not supported. Work on GenoM [9] is one of the closest to ours. It supports verification of schedulability and deadlock checking. Unlike RoboChart, GenoM is an executable language (potentially including C code) with limited support for abstractions.

The approach in [7] uses model checking to identify optimal configurations, but verification of behavioural properties is not the goal. Orccad [5] supports modelling, simulation, programming, and verification of timed behavioural properties. Verification is supported by translating models into formal languages like in our work. However, Orccad differs from RoboChart in its limited support for graphical modelling and granularity of its modelling elements.

To the best of our knowledge, the notations for robotics in the literature do not support modelling and verification of collections of robots. UML-like notations provide support for modelling of components. A distinguishing feature of RoboChart, though, is that it allows specifying a swarm configuration by relating meta (as opposed to concrete) robot instances via placeholders. Therefore, a model identifies the relevant communication patterns and specifies them as templates. The number of robots that form a concrete configuration can be a parameter of the more abstract configuration specification. As far as we are aware, all other existing graphical notations require that configurations are specified by relating concrete instances. This is the case, for instance, when we use Structure Diagrams to define a system configuration in terms of UML or SysML components. For every change in the number of components, the diagram needs to be revised. In RoboChart, we need to provide only a new value for a constant.

7 Conclusions

RoboChart supports modelling, verification, and simulation of robotic applications. Its concise design allows for the full specification of well-formedness conditions and semantics, as well as the implementation in the form of RoboTool. In this paper, we have described RoboChart support for explicit modelling of collections, and complex communication patterns within collections. The formal semantics of these facilities has been described and mechanised in RoboTool.

Currently, RoboChart is being extended with support for probabilistic modelling and verification, and a library of robotic platforms and common behaviours is under development. Furthermore, extensions to support modelling of the continuous aspects of the hardware and the environment are planned. All these extensions under development are useful also for modelling swarms.

Model checking is limited in the size of the models that it can handle. Our plan, especially for verification of swarms, is to explore compositional techniques for the efficient verification of CSP specifications [6], and semi-automatic verification using a CSP mechanisation in the theorem prover Isabelle [8].

The only form of communication available within a collection is perfect broadcast. We plan to provide: (1) a catalogue of types of communication media including mechanisms to model message loss and corruption; and (2) constructs and a library of models to support the specification of environments.

Acknowledgements. The work mentioned here is supported by the EPSRC grants EP/M025756/1 and EP/R025479/1, and by the Royal Academy of Engineering, and by INES, grants CNPq/465614/2014-0 and FACEPE/APQ/0388-1.03/14.

References

1. Broy, M., Cengarle, M.V., Rumpe, B.: Semantics of UML - towards a system model for UML: the state machine model. Technical report TUM-I0711, Institut für Informatik, Technische Universität München, February 2007
2. Davies, J., Crichton, C.: Concurrency and refinement in the unified modeling language. Formal Aspects Comput. **15**(2–3), 118–145 (2003)
3. Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., Ziane, M.: RobotML, a domain-specific language to design, simulate and deploy robotic applications. In: Noda, I., Ando, N., Brugali, D., Kuffner, J.J. (eds.) SIMPAR 2012. LNCS (LNAI), vol. 7628, pp. 149–160. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34327-8_16
4. Dixon, C., Winfield, A.F.T., Fisher, M., Zeng, C.: Towards temporal verification of swarm robotic systems. Robot. Auton. Syst. **60**(11), 1429–1441 (2012)
5. Espiau, B., Kapellos, K., Jourdan, M.: Formal verification in robotics: why and how? In: Giralt, G., Hirzinger, G. (eds.) Robotics Research, pp. 225–236. Springer, London (1996). https://doi.org/10.1007/978-1-4471-1021-7_26
6. Conserva Filho, M.S., Oliveira, M.V.M., Sampaio, A.C.A., Cavalcanti, A.L.C.: Compositional and local livelock analysis for CSP. Inf. Process. Lett. **133**, 21–25 (2018)
7. Fleurey, F., Solberg, A.: A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 606–621. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04425-0_47
8. Foster, S., Zeyda, F., Woodcock, J.: Isabelle/UTP: a mechanised theory engineering framework. In: Naumann, D. (ed.) UTP 2014. LNCS, vol. 8963, pp. 21–41. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14806-9_2
9. Foughali, M., Berthomieu, B., Dal Zilio, S., Ingrand, F., Mallet, A.: Model checking real-time properties on the functional layer of autonomous robots. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM 2016. LNCS, vol. 10009, pp. 383–399. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47846-3_24
10. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3—a modern refinement checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 187–201. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_13
11. Object Management Group. OMG: UML profile for MARTE, v1.0, November 2009. OMG Document Number: formal/(2009–11-02)
12. Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August 2011
13. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice-Hall, Upper Saddle River (1998)
14. Kuske, S., Gogolla, M., Kollmann, R., Kreowski, H.-J.: An integrated semantics for UML class, object and state diagrams based on graph transformation. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 11–28. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-47884-1_2
15. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A.L.C., Timmis, J.: Automatic property checking of robotic applications. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 3869–3876 (2017)
16. Nordmann, A., Hochgeschwender, N., Wigand, D., Wrede, S.: A survey on domain-specific modeling and languages in robotics. J. Softw. Eng. Robot. **7**(1), 75–99 (2016)

17. OMG: OMG systems modeling language (OMG SysML), Version 1.3 (2012)
18. Posse, E., Dingel, J.: An executable formal semantics for UML-RT. Softw. Syst. Model. **15**, 1–39 (2014)
19. Rasch, H., Wehrheim, H.: Checking consistency in UML diagrams: classes and state machines. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 229–243. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39958-2_16
20. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall Series in Computer Science. Prentice-Hall, Upper Saddle River (1998)
21. Schlegel, C., Hassler, T., Lotz, A., Steck, A.: Robotic software systems: from code-driven to model-driven designs. In: 14th International Conference on Advanced Robotics, pp. 1–8. IEEE (2009)
22. University of York: RoboChart Reference Manual. www.cs.york.ac.uk/circus/RoboCalc/robotool/

On the Industrial Uptake of Formal Methods in the Railway Domain

A Survey with Stakeholders

Davide Basile[1,3], Maurice H. ter Beek[1], Alessandro Fantechi[1,3], Stefania Gnesi[1], Franco Mazzanti[1], Andrea Piattino[2], Daniele Trentini[2], and Alessio Ferrari[1(✉)]

[1] ISTI–CNR, Pisa, Italy
{basile,terbeek,gnesi,mazzanti,alessio.ferrari}@isti.cnr.it
alessandro.fantechi@unifi.it
[2] SIRTI S.p.A., Genoa, Italy
{a.piattino,d.trentini}@sirti.it
[3] Università di Firenze, Florence, Italy

Abstract. The railway sector has seen a large number of successful applications of formal methods and tools. However, up-to-date, structured information about the industrial usage and needs related to formal tools in railways is limited. As a first step to address this, we present the results of a questionnaire submitted to 44 stakeholders with experience in the application of formal tools in railways. The questionnaire was oriented to gather information about industrial projects, and about the functional and quality features that a formal tool should have to be successfully applied in railways. The results show that the most used tools are, as expected, those of the B family, followed by an extensive list of about 40 tools, each one used by few respondents only, indicating a rich, yet scattered, landscape. The most desired features concern formal verification, maturity, learnability, quality of documentation, and ease of integration in a CENELEC process. This paper extends the body of knowledge on formal methods applications in the railway industry, and contributes with a ranked list of tool features considered relevant by railway stakeholders.

1 Introduction

The railway field is known for its robust safety requirements and its rigorous development processes. In fact, formal methods and tools have been widely applied to the development of railway systems during the last decades (cf., e.g., [1,2,4–7,9,11–17,21–24]) and the CENELEC EN 50128 standard for the development of software for railway control and protection systems mentions formal methods as highly recommended practices for SIL 3–4 platforms [8,10]. The extensive survey on formal methods applications by Woodcock et al. [25], which included a structured questionnaire submitted to the participants of 56 projects, also identified the transport domain, including railways, as the one

© Springer Nature Switzerland AG 2018

C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 20–29, 2018.
https://doi.org/10.1007/978-3-319-98938-9_2

in which the largest number of projects including applications of formal methods has been performed. Relevant examples are the usage of the B method for developing railway signalling systems in France, like, e.g., Line 14 of the Paris Métro and the driverless Paris Roissy Airport shuttle [1]. Another is the usage of Simulink/Stateflow for formal model-based development, code generation, model based-testing and abstract interpretation in the development of the Metrô Rio ATP system [11]. Many projects have been also carried out, often in collaboration with national railway companies, for the verification of interlocking systems [13,20–24].

Despite this long tradition and history, no universally accepted formal method or tool has emerged. Thus, on the one hand, railway companies wishing to introduce formal methods have little guidance for the selection of the most appropriate formal methods to use to develop their systems. On the other hand, tool vendors lack a clear reference concerning the features that are relevant for users of a tool in the railway domain. This paper aims to provide a first contribution to address these issues by presenting the results of a questionnaire submitted to experts in the theory and practice of formal methods in railways. The questionnaire's goal is to: (a) show the trends in the application of formal methods to railway systems, and (b) identify the most relevant features that a tool should support to be applied in railway systems' development.

This work is the first output of a larger endeavour that the authors are performing in the context of the ASTRail EU project[1] (SAtellite-based Signalling and Automation SysTems on Railways along with Formal Method and Moving Block Validation), funded by EU's Shift2Rail initiative[2]. A specific work stream of the project is concerned with an assessment of the suitability of formal methods in supporting the transition to the next generation of ERTMS/ETCS signalling systems [2–4]. The work stream's roadmap follows the two phases:

1. An *analysis* phase dedicated to survey, compare and evaluate the main formal methods and tools currently used in the railway industry.
2. An *application* phase in which selected formal methods are used to model and analyse two main goals of the project (moving block distancing and automatic driving) to validate that the methods not only guarantee safety, but also, more in general, the software's long-term reliability and availability.

The work presented in this paper is part of the analysis phase of ASTRail, in which the information retrieved with the questionnaire will be complemented with a systematic literature review and a systematic tool trial. Based on these tasks, we aim to complement the survey of Woodcock et al. [25] with a specific, in-depth focus on railway applications.

The paper is structured as follows: In Sect. 2, we provide information about the criteria used to define the questionnaire, and afterwards we present its results in Sect. 3. In Sect. 4, we provide conclusions and final remarks.

[1] http://astrail.eu.
[2] http://shift2rail.org.

2 Questionnaire Definition

For the nontrivial task of obtaining a significative amount of data from industrial stakeholders, a survey was carried out by means of a structured questionnaire, submitted to the participants of the recent RSSRail'17 conference[3]. This venue is attended by academics and practitioners interested in applying formal methods in railways, and as such a promising source for a population sample that might be able to provide a well-informed judgement.

The goal of the questionnaire was to: (a) identify the current uptake of formal and semi-formal methods and tools in the railway sector; (b) identify the features, in terms of functional and quality aspects, that are considered more relevant for the application of a certain formal tool in the development of railway products. The questionnaire was designed to be easy to understand by the target group, involving academics and practitioners, and to be filled within five minutes, to limit the amount of time required for the people surveyed, and possibly increase the number of respondents. The design of the questionnaire was performed by the authors of the current paper, who include both academics with expertise in formal methods applied to railways and practitioners from railway industry. For the questions concerning the relevance of the tool features (cf. Sect. 3.3), a two-hour brainstorming session based on the KJ-method [18] was organised to identify possibly relevant features. The questionnaire was tested and validated with industrial partners of the ASTRail consortium for clarity and the time required. An online version of the questionnaire, which the reader can refer to have a clear view of the proposed questions, can be found at the following link: https://goo.gl/forms/4b9wSTJAMOK7VghW2.

3 Results of the Questionnaire

In the following sections, we report and interpret the results that we obtained.

3.1 Affiliations and Experience

The first part of the questionnaire was dedicated to identify the respondents in terms of affiliation and experience in railways and formal/semi-formal methods and tools. The 44 respondents are balanced between academics (50%) and practitioners (50%, of which 47.7% from railway companies and 2.3% from aerospace and defense). A large percentage of respondents has several years of experience in railways (68% more than 3 years and 39% more than 10 years) and in formal methods (75% more than 3 years, 52% more than 10 years), and this confirms that our sample can provide informed opinions on the proposed questions[4].

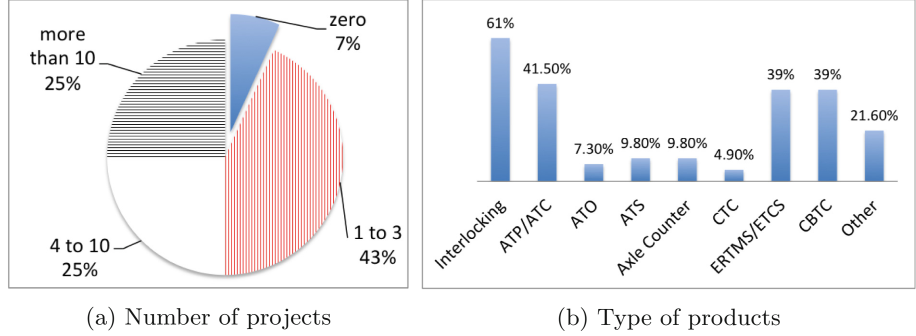
[3] http://conferences.ncl.ac.uk/rssrail/.

[4] We did not weigh the results based on the declared experience of the respondents, because we wanted to give equal importance to their different answers, regardless of the specific experience.

3.2 Usage of Formal Methods in Railway Sector

The second part of the questionnaire was oriented to have an insight on the usage of formal/semi-formal methods and tools in railways.

Projects. We asked in how many *industrial* railway projects the respondents, or their teams, have used formal/semi-formal methods and tools. Since the respondents included also academics, we expected that the industrial projects in which they were involved were mainly technology transfer projects with companies. Figure 1a shows that only 7% of the respondents—or their teams—did not have any industrial experience in the application of formal methods in railways[5].



(a) Number of projects (b) Type of products

Fig. 1. Usage of formal methods in the railway sector

Products. Figure 1b shows the main types of products developed with the support of formal methods. The cited systems include an extensive range of signalling systems and components. The majority of the respondents applied formal methods to interlocking systems (61% of the respondents[6]), but also automatic train protection/automatic train control (ATP/ATC) distancing systems (41.5%), especially in their standardised form for main lines (ERTMS/ETCS, 39%) or for metro lines (CBTC, 39%) play a major role. Automatic train operation (ATO), automatic train supervision (ATS), axle counter systems and centralised traffic control (CTC) are also mentioned. This prominence of in particular interlocking and ATP/ATC systems is in line with the formal methods literature, for which these types of systems are traditional applications [9].

Phases. With the aim of estimating the degree of integration of formal methods in software engineering practice, respondents were asked to indicate the phase of the development process in which formal methods are applied (cf. Fig. 2). We see that all phases have been selected by at least one of the respondents, highlighting the potential pervasiveness of formal methods within the development process.

[5] When present, the subsequent answers of these respondents were discarded from our statistics, since they were considered outliers with respect to our population sample.

[6] For this and subsequent questions, respondents could select more than one answer.

Most of the respondents (73.8%) used them for specification and formal verification. Also analysis of specifications (50%) and simulation (40.5%) appear to be common, and a non-negligible amount of respondents (31%) used formal methods also within model-based testing and code generation contexts. Less common (7.1%) is their application to the static analysis of the source code.

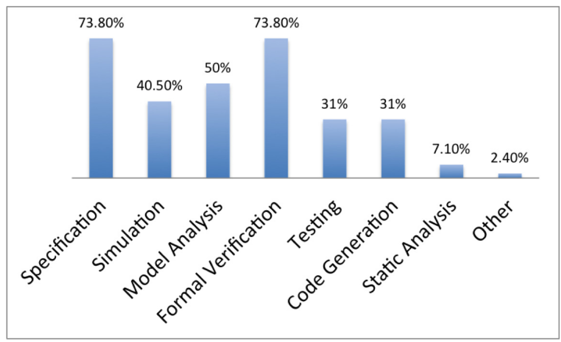


Fig. 2. Phase of the process in which formal methods are applied

Tools. The respondents were also asked to list the tools they have used in the context of their projects, and, in this case, we believe it is interesting to separate the results of industrial respondents from those of academics. In Fig. 3, we can see that the large majority of industrial and academic respondents mentioned tools belonging to the B method family (e.g. B, ProB, AtelierB, EventB, RODIN). The relationship between the B method and the railway sector is well established: as Sun [19] puts it, "the B proved models are considered *safe* in French industry." Actually, there are only slightly more industrial users than academic users in our sample, but we recall that the academic users were asked to report on their collaborative projects with industry. Other methods and tools mentioned by both groups are the Matlab toolsuite—including Simulink and Stateflow—SCADE, Petri nets/CPN tools and Monte Carlo Simulation: the overlapping between tools used in industry and in academia is actually limited to these five elements. Industrial users named a few other tools as well, whereas a large list of other tools has been named by academics, with popular model checkers like NuSMV and SPIN leading this list. An interpretation of this can be that a frequent pattern of collaboration between academia and industry includes the academic support in adopting advanced formal verification techniques inside a collaborative project.

3.3 Feature Relevance

The final part of the questionnaire was dedicated to identify the most relevant features that a formal/semi-formal tool should have to be used in the railway industry. Features are partitioned into supported functional and quality aspects. We asked to check at most three relevant functional features, among the seven listed, and at most five relevant quality aspects, among the sixteen listed.

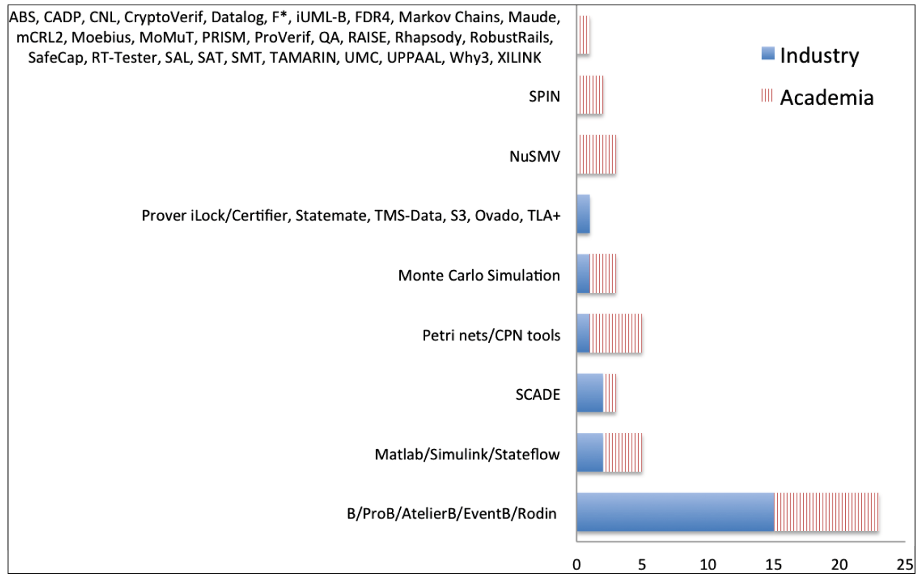


Fig. 3. Tools cited in the questionnaire

Functional Features. Figure 4 shows the results for the most relevant functional features. All the listed features are considered relevant by at least one of the respondents. The functional features that are considered most relevant by the majority of the respondents are formal verification (86.4% of the respondents), followed by modelling—graphical or textual—(72.7%). These traditional functional features of formal tools are followed by simulation (30%) and traceability (27.3%). Indeed, simulation (often in the form of animation of a graphical specification) is needed for a quick check of the behaviour of a model; traceability

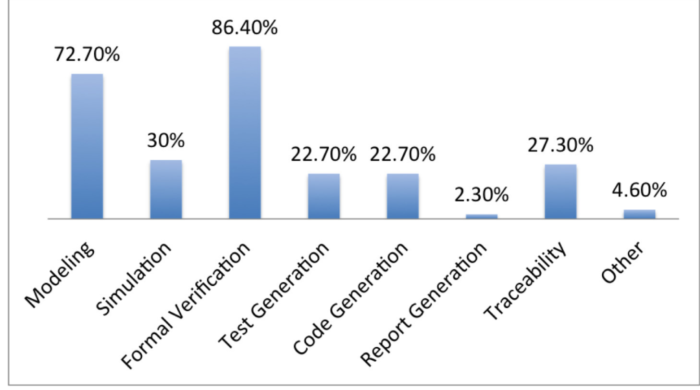


Fig. 4. The most relevant functional features a (semi-)formal tool should support

between the artefacts of the software development (requirements to/from models, models to/from code, etc.) is mandatorily required by the main guidelines for the development of safety-critical systems. Functional features, such as test generation and code generation, related to later activities of the development process, are also considered relevant by a non-negligible amount of respondents (22.7%). These numbers suggest that formal tools are seen to play a role mostly in the early phases of the development process, for specification and formal verification. These are also the phases in which formal methods cannot be substituted by any other means—while this may happen in testing, code development and tracing.

Quality Aspects. Figure 5, finally, reports the most relevant quality aspects and, also in this case, all the listed answers were checked by at least one of the respondents. The maturity of the tool (stability and industry readiness) is considered to be among the most relevant quality aspects by 75% of the respondents, followed by learnability by a railway software developer (45.5%), quality of documentation (43.2%) and ease of integration in the CENELEC process (36.4%). Overall, the most relevant quality aspects are associated to the usability of the tool. Less relevant are deployment aspects, such as platforms supported (9.1%) and flexible license management (11.4%). Interestingly, also the low cost of the tool (13.6%) appears to be a not extremely relevant feature. This is a reasonable finding. Indeed, the development and certification cost of railway products is high and, hence, if a company expects to reduce these costs through a formal tool, it can certainly tolerate the investment on the tool.

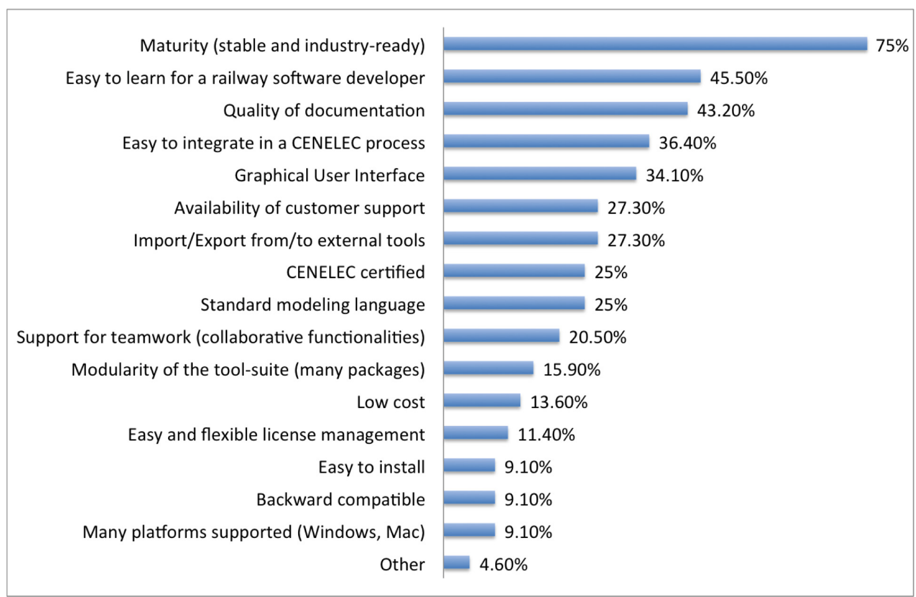


Fig. 5. The most relevant quality aspects a (semi-)formal tool should have

3.4 Threats to Validity

Concerning construct and internal validity, the questions defined and the options proposed as answers may be incomplete to identify practical uses of tools, and desired features. Furthermore, the respondents may have misunderstood the meaning of the questions. To mitigate these threats, the questions were designed and tested in collaboration between academic and industrial partners.

Concerning statistical conclusion validity, we do not have an estimate of the whole population of subjects applying formal methods in railways, and our sample was limited to the participants of RSSRail. However, assuming that the population of persons applying formal methods in railways is $1,000$, our results on a sample of 44 persons are valid for a confidence level of 85% and margin of error of 10.5%. While higher values are normally targeted in qualitative research, the answers to the questionnaire show that the sample is made of high-quality (i.e. informed) respondents, which increases the reliability of our results. However, we cannot exclude that important industrial applications of formal methods are not public, and people working on them may not attend conferences like RSSRail, also for confidentiality policies.

4 Conclusion

Formal methods and tools have been applied quite extensively in specific industrial domains, especially those in which safety-critical software is produced, either in pilot projects or in daily production. On the other hand, industry often confronts itself with the choice among a large variety of techniques and tools, with little help for selecting the ones that better fit their needs. Within the H2020 ASTRail project, the authors are working on providing information to guide railway practitioners interested in the adoption of formal methods.

To this end, we performed the questionnaire presented in this paper and we are working on a literature survey on formal methods for railways, as well as on a systematic tool evaluation (cf. [14,16] for preliminary comparisons of formal modelling and verification frameworks). The current work provides preliminary information on the industrial uptake of formal methods in railways. The results show that, although the B method appears to be the one that is mostly used in the railway industry, several other tools have been used, and some of them are not even considered by the academics that were part of the respondents. Furthermore, we observed that industrial needs concerning formal tools are mostly related to usability features, such as maturity of the tools, learnability, and quality of documentation. Interestingly, the cost of the tools is not a highly relevant issue, suggesting that industry appears to be available to invest in formal tools, if these guarantee a process cost reduction and the expected safety assurance.

Acknowledgements. This work has been partially funded by the ASTRail project. This project received funding from the Shift2Rail Joint Undertaking under the European Union's Horizon 2020 research and innovation programme under grant agreement No. 777561.

References

1. Abrial, J.R.: Formal methods: theory becoming practice. J. Univ. Comput. Sci. **13**(5), 619–628 (2007). https://doi.org/10.3217/jucs-013-05-0619
2. Basile, D., ter Beek, M.H., Ciancia, V.: Statistical model checking of a moving block railway signalling scenario with Uppaal SMC. In: Margaria, T., Steffen, B. (eds.) Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018). LNCS. Springer, Heidelberg (2018, to appear)
3. ter Beek, M.H., Fantechi, A., Ferrari, A., Gnesi, S., Scopigno, R.: Formal methods for the railway sector. ERCIM News **112**, 44–45 (2018). https://ercim-news.ercim.eu/en112/r-i/formal-methods-for-the-railway-sector
4. ter Beek, M.H., Fantechi, A., Gnesi, S.: Product line models of large cyber-physical systems: the case of ERTMS/ETCS. In: Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC 2018). ACM (2018). https://doi.org/10.1145/3233027.3233046
5. ter Beek, M.H., Gnesi, S., Knapp, A.: Formal methods for transport systems. Int. J. Softw. Tools Technol. Transf. **20**(3), 237–241 (2018). https://doi.org/10.1007/s10009-018-0487-4
6. Bjørner, D.: New results and trends in formal techniques and tools for the development of software for transportation systems – a review. In: Tarnai, G., Schnieder, E. (eds.) Proceedings of the 4th Symposium on Formal Methods for Railway Operation and Control Systems (FORMS 2003). L'Harmattan (2003)
7. Boulanger, J.L. (ed.): Formal Methods Applied to Industrial Complex Systems - Implementation of the B Method. Wiley, Hoboken (2014). https://doi.org/10.1002/9781119002727
8. European Committee for Electrotechnical Standardization: CENELEC EN 50128 – railway applications - communication, signalling and processing systems - software for railway control and protection systems, 1 June 2011. https://standards.globalspec.com/std/1678027/cenelec-en-50128
9. Fantechi, A.: Twenty-five years of formal methods and railways: what next? In: Counsell, S., Núñez, M. (eds.) SEFM 2013. LNCS, vol. 8368, pp. 167–183. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05032-4_13
10. Fantechi, A., Ferrari, A., Gnesi, S.: Formal methods and safety certification: challenges in the railways domain. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 261–265. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_18
11. Ferrari, A., Fantechi, A., Magnani, G., Grasso, D., Tempestini, M.: The Metrô Rio case study. Sci. Comput. Program. **78**(7), 828–842 (2013). https://doi.org/10.1016/j.scico.2012.04.003
12. Flammini, F. (ed.): Railway Safety, Reliability, and Security: Technologies and Systems Engineering. IGI Global, Hershey (2012). https://doi.org/10.4018/978-1-4666-1643-1
13. James, P., Moller, F., Nguyen, H.N., Roggenbach, M., Schneider, S., Treharne, H.: Techniques for modelling and verifying railway interlockings. Int. J. Softw. Tools Technol. Transf. **16**, 685–711 (2014). https://doi.org/10.1007/s10009-014-0304-7

14. Mazzanti, F., Ferrari, A.: Ten diverse formal models for a CBTC automatic train supervision system. In: Gallagher, J.P., van Glabbeek, R., Serwe, W. (eds.) Proceedings of the 3rd Workshop on Models for Formal Analysis of Real Systems and the 6th International Workshop on Verification and Program Transformation (MARS/VPT 2018). Electronic Proceedings in Theoretical Computer Science, vol. 268, pp. 104–149 (2018). https://doi.org/10.4204/EPTCS.268.4
15. Mazzanti, F., Ferrari, A., Spagnolo, G.O.: Towards formal methods diversity in railways: an experience report with seven frameworks. Int. J. Softw. Tools Technol. Transf. **20**(3), 263–288 (2018). https://doi.org/10.1007/s10009-018-0488-3
16. Mazzanti, F., Spagnolo, G.O., Della Longa, S., Ferrari, A.: Deadlock avoidance in train scheduling: a model checking approach. In: Lang, F., Flammini, F. (eds.) FMICS 2014. LNCS, vol. 8718, pp. 109–123. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10702-8_8
17. Moller, F., Nguyen, H.N., Roggenbach, M., Schneider, S., Treharne, H.: Defining and model checking abstractions of complex railway models using CSP||B. In: Biere, A., Nahir, A., Vos, T. (eds.) HVC 2012. LNCS, vol. 7857, pp. 193–208. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39611-3_20
18. Scupin, R.: The KJ method: a technique for analyzing data derived from Japanese ethnology. Hum. Organ. **56**(2), 233–237 (1997). https://doi.org/10.17730/humo.56.2.x335923511444655
19. Sun, P.: Model based system engineering for safety of railway critical systems. Ph.D. thesis, Ecole Centrale de Lille (2015). https://tel.archives-ouvertes.fr/tel-01293395
20. Vanit-Anunchai, S.: Modelling and simulating a Thai railway signalling system using coloured Petri Nets. Int. J. Softw. Tools Technol. Transf. **20**(3), 243–262 (2018). https://doi.org/10.1007/s10009-018-0482-9
21. Vu, L.H., Haxthausen, A.E., Peleska, J.: Formal modelling and verification of interlocking systems featuring sequential release. Sci. Comput. Program. **133**, 91–115 (2017). https://doi.org/10.1016/j.scico.2016.05.010
22. Winter, K.: Model checking railway interlocking systems. In: Oudshoorn, M.J. (ed.) Proceedings of the 25th Australasian Conference on Computer Science (ACSC 2002). Conferences in Research and Practice in Information Technology, vol. 4, pp. 303–310. Australian Computer Society (2002). http://crpit.com/confpapers/CRPITV4Winter.pdf
23. Winter, K., Johnston, W., Robinson, P., Strooper, P., van den Berg, L.: Tool support for checking railway interlocking designs. In: Cant, T. (ed.) Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software (SCS 2005). Conferences in Research and Practice in Information Technology, vol. 55, pp. 101–107. Australian Computer Society (2006). http://crpit.com/confpapers/CRPITV55Winter.pdf
24. Winter, K., Robinson, N.J.: Modelling large railway interlockings and model checking small ones. In: Oudshoorn, M.J. (ed.) Proceedings of the 26th Australasian Computer Science Conference (ACSC 2003). Conferences in Research and Practice in Information Technology, vol. 16, pp. 309–316. Australian Computer Society (2003). http://crpit.com/confpapers/CRPITV16Winter.pdf
25. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.S.: Formal methods: practice and experience. ACM Comput. Surv. **41**(4), 19:1–19:36 (2009). https://doi.org/10.1145/1592434.1592436

Reasoning About JML: Differences Between KeY and OpenJML

Jan Boerman, Marieke Huisman, and Sebastiaan Joosten(✉)

University of Twente, Enschede, The Netherlands
j.g.j.boerman@student.utwente.nl,
{m.huisman,s.j.c.joosten}@utwente.nl

Abstract. To increase the impact and capabilities of formal verification, it should be possible to apply different verification techniques on the same specification. However, this can only be achieved if verification tools agree on the syntax and underlying semantics of the specification language and unfortunately, in practice, this is often not the case.

In this paper, we concentrate on one particular example, namely Java programs annotated with JML, and we present a case study in understanding differences in the treatment of these specifications. Concretely, we take a collection of JML-annotated programs, that we tried to reverify using KeY and OpenJML. This effort led to a list of syntactical and semantical differences in the JML support between KeY and OpenJML. We discuss these differences, and then derive some general principles on how to improve interoperability between verification tools, based on the experiences from this case study.

Keywords: Java Modeling Language · Static verification
OpenJML · KeY

1 Introduction

As a society, we increasingly rely on digital technology driven by software, and therefore we need formal techniques to provide guarantees about the quality and reliability of software. There is a wide plethora of tools and techniques available that contribute to this. However, all these tools and techniques have their own strong and weak points, and in order to increase impact and usability of formal techniques, we will need to find ways to combine them.

Unfortunately, combining tools is often not that straightforward, because even though in principle they implement the same specification language, they differ in the details, both in the syntax and the semantics.

This paper presents a case study to investigate the chances and difficulties for tool interoperability in one specific setting, namely Java programs annotated with JML (the Java Modeling Language) [18]. A wide range of different tools exist that take JML-annotated Java programs as input, and implement checks

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 30–46, 2018.
https://doi.org/10.1007/978-3-319-98938-9_3

to establish whether the program behaves as specified by the annotations. Moreover, there are even tools that try to automatically come up with a suitable JML specification for a given Java program, see e.g. [5] for an overview of JML tools. Despite efforts to agree on a common core language for JML, in practice there are still a lot of differences between all these tools, both syntactically and semantically. To improve this situation it is important to obtain a precise understanding of these differences.

In this paper, we consider two tools that take JML-annotated Java programs as input, and then apply static verification support, namely KeY [1] and OpenJML [9]. KeY is an interactive program verifier, based on dynamic logic, which typically makes it suitable for the verification of complex methods, because the user can incrementally build the proof. In constrast, OpenJML works fully automatically: from the annotated program, verification conditions are generated and sent to a first-order prover. This makes verification very fast for typical boilerplate methods (getters and setters) where the correct specifications can be given directly, but is less suited for incremental development of a specification. Thus, there is a high potential to increase verification efficiency if a user can smoothly switch between OpenJML and KeY during the verification process.

In order to investigate whether this switching could indeed be a smooth process, we took several sources of annotated examples that we tried to reverify in KeY and OpenJML. These include the examples from the KeY website (from www.key-project.org), as well as some hand-crafted examples that came up in the investigation of which keywords are supported and which are not. This resulted in a list of syntactical and semantical differences between the tools that should be addressed, or at least made explicit, so a user knows where to expect the differences. For all these examples of interest, we developed a minimal variant of the program in order to illustrate the issue in isolation. We do not believe this list of differences to be exhaustive, however we believe that it nicely illustrates the typical issues that have to be understood in order to enable tool interoperability.

It is important to stress that with this paper, we do not wish to argue for one tool over the other; we only would like to make it clear that there are differences in their behaviour, which can be unexpected for a tool user. We hope that the comparison helps eventually to make it easier to switch between different verification tools. Importantly, the authors of this paper have not been involved in the development of OpenJML or KeY, but they have a thorough experience with JML annotations, and are teaching JML-style specifications to Bachelor and Master students. Therefore, even though we try to find reasons to explain the differences in behaviour of KeY and OpenJML, the real reasons might be different. And of course, the differences that we identify between KeY and OpenJML are not just a warning for users of these tools: they are also an invitation to developers of these tools and developers of the JML standard to come to an agreement on a unified semantics for JML.

Finally, the last part of this paper tries to derive some general lessons from the experiences obtained with this case study in tool comparison. How can we

avoid having to do such tool comparisons between all possible combinations of verification tools? How can we improve the situation upfront, in such a way that the potential differences between tools become more apparent? Or should we simply aim for a situation where all differences between tools are resolved? We hope that these lessons learned will inspire the community to look more closely at tool interoperability and the necessary steps to achieve this.

Contributions. The main contribution of this case study is that it presents a collection of small JML-annotated programs for which verification behaves differently in KeY and OpenJML. Each program is designed to be minimal, and to illustrate a single issue in isolation, which makes them easy to understand and analyze for users and developers of tools and the JML language.

These examples should help users of KeY and OpenJML to better understand how easy or difficult it will be to switch between the two tools. They should also help KeY and OpenJML developers to better understand the strengths and weaknesses of their own tool, in comparison to the other tool. Finally, they should help developers of the JML standard to know what parts of the standard need to be clarified and what needs to be accounted for.

Scope of the Examples. Even though this paper may not cover all differences between KeY and OpenJML, we tried to choose our example set in a systematic manner. First, we systematically went through the documentation of JML [18] and tried to verify 'firsttouch' features individually. Second, we took the examples from the KeY website, and tried to verify these both in KeY and OpenJML. Third, we looked at the self-reported differences from the JML standard [12,13], to come up with examples there. Finally, we created our own small examples in an ad-hoc fashion if a suspicion arose of a possible difference, while working with the examples stated above.

Related Work. There are two webpages that list differences between JML and OpenJML [13], and differences between JML and KeY [12], respectively. However, these webpages focus on keyword support only – we will discuss in Sect. 3 – and are very brief.

Another comparison is made in the appendix of the JML standard itself [18, Sect. D], namely between JML and the specification language of ESC/Java (Extended Static Checker for Java) [19], which is a Java annotation language very similar to JML. Unfortunately, this comparison is outdated: it seems not to have been updated since 2003. And in particular, it does not contain a comparison between JML and the successors of ESC/Java, ESC/Java2, and later OpenJML. Fortunately, such a comparison is made upon the introduction of ESC/Java2 by the authors themselves [8].

We are not aware of many similar comparisons between tools. One of the authors of this paper has made a comparison between the interactive theorem provers PVS and Isabelle/HOL [10]. A comparison between several tools (including KeY and OpenJML) is made by Thüm et al. [21] for the purpose of aggregating the tools into a single model checking and theorem proving, focusing mainly

on similarities. What is new in this case study is that we take a collection of externally developed examples, and base our experiences on this.

After reading an earlier version of this paper, David Cok, the main author of OpenJML, has given us some feedback through private communication. Where relevant, we included his remarks (clearly indicated as being his).

Overview of This Paper. The remainder of this paper is organised as follows. Section 2 gives a short introduction to JML and briefly describes OpenJML and KeY. Section 3 then discusses several syntactical differences between JML as supported by KeY and by OpenJML. Then, Sect. 4 continues with the semantical differences we have observed. Finally, Sect. 5 draws some general lessons from our experiences.

2 Background: Static Verification with KeY and OpenJML

One way to verify correctness of Java code is by adding contracts to methods, and verifying those contracts. The standard in which to describe contracts for Java is JML [16–18]. A JML method contract essentially consists of two parts: one is called 'requires' and indicates the assumptions under which a method is called; the other is called 'ensures' and indicates the guarantees that the method gives. A small example is given in Fig. 1. This specification states that the method should only be called with arguments `d` and `v` being strictly positive, and that it will return a value in the interval `[v - d, v]`. JML allows several variants on this: depending on whether or not a method is expected to always terminate, and whether or not it could throw exceptions, different kinds of contracts can be chosen.

Verifying JML contracts can be done in several ways. Popular approaches include runtime verification and static verification. Runtime verification will check validity of contracts during the execution of the program, while static verification will try to establish statically, without executing the program, whether the contracts are always respected. Runtime and static verification are orthogonal approaches: runtime verification finds errors when they happen, but does not verify the correctness of all possible programs, while static verification aims to prove correctness of all executions, but might indicate errors that will never happen during an execution. In particular, if a program does not have sufficient annotations, static verification might fail, even though the program is correct. However, if there is an issue with the program, it will be reported.

In static verification, typically only the contract of a method is used to reason about invocations of that method: The 'requires' part of a contract is the precondition, to be proven in the state before a method is called. Then the 'ensures' part of that contract, which is its postcondition, can be assumed in the state after its call. This makes static verification with JML highly modular: implementations can be changed freely as long as the contract remains provable, and the rest of the verification effort will remain valid. For a detailed analysis on

```
1 class SimpleContract {
2
3     /*@ requires d > 0 && v > 0;
4       @ ensures v - d <= \result && \result <= v;
5       @*/
6     public static int round(int v, int d) {
7       return v - (v % d);
8     }
9 }
```

Fig. 1. Method with a contract

the benefits of using contracts for verification (over inlining), see the work by Knüppel et al. [15], which includes experiments in KeY.

A static verification tool transforms the specified program and its contracts into proof obligations. How the translation is done depends on the tool used. These proof obligations are then checked in some way, which again depends on the verification tool used. Not only the contract specification is translated during such a transformation, but certain implicit language rules are as well. In Fig. 1, there is a potential division-by-zero-error on line 7. However, the precondition of the method suffices to show that this division-by-zero-error will not occur.

OpenJML. OpenJML [9] is developed as the successor of ESC/Java2 and the runtime verification tool suite for JML [7]. For static verification, it transforms a JML-annotated program into a static single assignment form, and then generates first-order logic verification conditions from this transformed program. This output format is suitable for a satisfiability modulo theory (SMT) solver. As such, it is given as input to an SMT solver, which by default is Z3 [20].

OpenJML can be used by invoking it from the command line, or as an Eclipse plugin. OpenJML can do run time verification as well as static verification. Throughout this case study, we invoke OpenJML from the command line with `-esc` plus a file name, to do static verification. The result can be a set of warnings and errors, or if the program has no issues, nothing at all. Indeed, for the program in Fig. 1, no output is given, indicating the program is correct. If verification does not go through, the warning or error points to the places in the program that are causing the issue. For instance, changing line 7 to **`return`** `d;` will give a postcondition violation error that points to line 4, as well as to line 7.

David Cok let us know that OpenJML also has an IDE in which counterexamples can be explored. We did not try the IDE for this work.

KeY. The KeY project positions itself as a portfolio of tools for program verification [1]. It has a tool for static program verification, allows test generation based on contracts, and has an Eclipse plugin for symbolic debugging. All of these tools are built on a common code base that includes an interactive prover and a symbolic evaluator of programs. KeY is based on dynamic logic, shared

by the symbolic evaluation and the interactive prover, into which programs and annotations are translated.

KeY allows additional axioms and strategies to be added to its prover through a feature called 'add user defined taclets'. Even the assumptions made in KeY's built-in taclets can be tweaked: there is a setting called 'taclet options' that allow us to determine how certain Java commands are treated. For instance, one can tell KeY to ignore assert commands to mimic the behavior of when the Java Virtual Machine (JVM) is called in the same way. Alternatively, one can tell KeY to treat failing assert commands as runtime exceptions, or to generate proof requirements that ensure that they hold. Proving that assertions hold means that the JVM will not report any assertion failures, regardless of whether assertion checking is enabled in the JVM. KeY allows us to save proofs, both completed and incomplete proofs. For this case study, we refrained from using any user defined taclets.

In the tool KeY, one opens a directory from a GUI. The tool then lists all proof obligations for all JML annotated files in that directory. When one is selected, one can interactively create a proof for it. There is also an automatic option. Using the automatic option on the program in Fig. 1 produces a proof with two open subgoals: One where a proof of `jmod(v, d) <= d` is required, and another where `jmod(v, d) < 0` is assumed, where `jmod` refers to the built-in function of the JVM that computes the modulo, which arises as a translation of `%`. KeY makes no assumptions about the JVM's implementation of `jmod`, so we cannot complete the proof. This does not depend on modes of taclets (like whether or not arithmetic is verified with overflow checks). We could, however, proceed by adding these assumptions to KeY manually by adding taclets, and in this way complete the proof.

3 Syntactical Differences

This section describes differences between KeY and OpenJML for which a syntactic criterion can be given. We first discuss differences in the parts of the JML standard that are covered by OpenJML and KeY. Then, we continue with the extensions to the JML standard offered by either KeY or OpenJML. Finally, we also briefly discuss what could be done to decrease the syntactical gaps. Table 1 summarizes the discussion in this section, and gives an overview which keywords are supported by which tool (and by the JML standards). This table does not show what is supported by the parser built into each tool, but rather by the tool when performing static verification. We have also omitted keywords for which we could not find clear differences, such as **`forall`**.

Covered JML Subset. For most syntactical differences, the JML standard seems to be a driving force: both KeY and OpenJML implementers aim to let JML keywords behave as described in the JML manual. The developers are generally aware of the syntactical differences: both the KeY website and the OpenJML website feature lists of differences and similarities between the tool and the JML

Table 1. Syntactic elements evaluated, and support in JML, OpenJML and KeY

Keywords	JML	KeY	OpenJML
\sum \product \num_of	Yes	Yes	No
\strictly_nothing strictly_pure	No	Yes	No
\not_assigned	Yes	No	No
\bSum \bProduct	No	Yes	No
\locset \intersect \set_union	No	Yes	No
\distinct	No	No	Yes
\index	No	Yes	Yes
Certain Java arithmetic: % ^	Via Java	No	Yes

standard [12,13]. Some JML keywords that KeY supports, but OpenJML does not are: **\sum**, **\product** and **\num_of**.

Non-JML Extensions. Other syntactic differences come from non-JML extensions to KeY. KeY has greater flexibility than OpenJML in indicating that certain variables may not be assigned: in OpenJML, like KeY, one can state for which variables the value may change through the execution of the method. All other variables should never be assigned. In KeY, however, one by default states that other variables will eventually return to their original value. The difference can be useful in concurrent programs (although neither KeY nor OpenJML currently support the verification of such programs). In KeY, within an **assignable** clause, the **\strictly_nothing** keyword indicates that no global variables may be assigned, even if they are eventually restored to their old value. Similarly, **strictly_pure** indicates the same thing. These keywords are not part of the JML standard, they implement what according to the JML standard should be the behaviour of **\nothing** and **pure** respectively. OpenJML does not offer a choice on how **assignable** is interpreted, but it does implement the JML standard by default. The JML keyword **\not_assigned**, which also serves this purpose in the JML standard, is not supported by OpenJML or KeY.

In conditions, the keywords **\bsum** and **\bProduct** are used as a bounded verifier-friendly version of **\sum** and **\Product**, respectively, which gets the range over which the sum or product is calculated as two integers.

For modeling the heap, KeY introduces the **\locset** type, as well as set operations like **\intersect**, **\set_minus** and **\intersect**, **\set_union**. Reasoning about the heap was introduced to support dynamic framing in KeY [2,22].

Like KeY, OpenJML has some non-JML extensions that are not supported by KeY. For conveniently writing that every pair in a set of variables is distinct, OpenJML writes **\distinct**, which can be considerably shorter than using != pairwise for large sets of variables.

Interestingly, the keyword **\index** is supported in both OpenJML and KeY, even though this keyword is not part of the JML standard. Within an enhanced for-loop (a for-each loop), **\index** is used to indicate the current index. The

\index keyword was discussed at a JML workshop, and may become part of JML as the **\count** keyword.

We did not focus on finding out which subset of Java is supported by KeY or OpenJML. However, as mentioned in the introduction, Fig. 1 did not automatically verify in KeY while it did verify in OpenJML. This seems to happen because KeY does not have a built-in axiomatization of `%` by default. Similarly, KeY could not verify any properties about the bit-wise xor, or ^ in Java, while OpenJML could. In contrast to OpenJML, KeY does not allow the use of Java generics, although the KeY website claims that these can be removed statically via an Eclipse plugin (which we did not test).

Reducing the Gap. The syntactical differences between KeY and OpenJML can be a nuisance for someone trying to use different tools for different parts of the same specification, and therefore these differences should be clearly documented and avoided as much as possible.

We recommend that KeY supports the **\not_assigned** keyword, and deprecates **\strictly_nothing** and **strictly_pure**. With the exception of **\locset** and the corresponding set operations, all keywords can be expressed in standard JML. For the use of **\locset**, it is worth considering adding this to the JML standard. We also recommend to add **\index** to the JML standard.

At some point, there has been a proposal to add markers to annotations, to indicate that they were tool-specific, because KeY would require different in-code annotations than e.g. OpenJML. One could imagine that annotations that use `/*KEY@ ....*/` as surrounding comments are considered only by KeY. This idea was introduced during a JML workshop, and is now supported by at least OpenJML through the markers `RAC`, `ESC`, and `OPENJML`.

4 Semantical Differences

The previous section discussed syntactical differences between the JML specifications supported by OpenJML and KeY. However, even more important are semantical differences, where the specifications are the same (maybe modulo syntactical differences), but the behaviour of the tools is different.

This section provides a list of such differences. We do not believe that this list is exhaustive, but we believe it gives a good impression of the semantical differences in tool behaviour that one should be aware of. In fact, we are not sure whether it is possible to give a fully exhaustive list of such differences, but we believe that understanding and discussing the differences is important for a better interoperability between tools.

The sources of the differences that we list here can vary: sometimes they are caused by the underlying prover technology (or might even be caused by a bug in the underlying solver), but they can also be related to a different semantical interpretation of the Java or JML semantics.

Compiler Checks. A difference that stands out immediately is that KeY does not require a class to be compilable, while OpenJML does require this. David Cok told us this is due to OpenJML's use of OpenJDK to produce ASTs.

The KeY approach provides flexibility, and has several advantages:

– it makes it possible to verify classes in isolation, without considering the complete hierarchy of all classes surrounding this class, and
– it is possibly to quickly copy the class that is being verified into a different file, without having to change the class name accordingly.

However, the disadvantage and major risk is that one might spend a lot of time on the verification of a non-compilable program (and this time might thus be completely wasted). The KeY approach thus requires more discipline from the users to make sure that they are indeed working on a correct Java file.

In contrast, OpenJML builds this check in, and thus immediately identifies program errors, but does not make it easy to verify single classes in isolation. As a result, in OpenJML one often has to spend a lot of time on stripping irrelevant imports, function calls etc., in order to make the tool actually check some specification.

Visibility Checks. Related is that KeY does not do visibility checks on fields and methods. In particular, it does not do these checks in the specifications. As a result, KeY does not complain if a publicly visible specification uses private variables. For example, in Fig. 2 the KeY-verified example uses private fields in the public method specification: both the private variable `a` and the private method `ReturnFive` occur in the **`ensures`** statement. In contrast, OpenJML immediately reports all visibility issues in this specification.

```
1  public class InitPrivateToPublic {
2    private int a;
3    /*@ ensures a == returnFive();
4      @*/
5    public InitPrivateToPublic() {
6       a = returnFive();
7    }
8
9    /*@ ensures \result == 5; @*/
10   private /*@ pure @*/ int returnFive(){
11     return 5;
12   }
13 }
```

Fig. 2. Publicly visible specification with private variables

This lack of visibility checking in KeY is in violation with the JML standard [18, Sect. 2.4], and we believe that this is an omission in the KeY implementation, because it breaks the standard rules of encapsulating an object's

internal state. Moreover, a simple solution is available by declaring the variable **spec_public**, which implicitly declares a model variable that abstracts from this internal state (and thus, if the internal state is changed, only the relation between the model variable and the internal state has to be adapted, but the public method specifications do not change).

```
1  public class InitPublic {
2     private /*@ spec_public @*/ int a;
3     /*@ public normal_behavior
4       @ ensures a == 5;
5       @*/
6     public InitPublic() {
7        a = returnFive();
8     }
9
10    private /*@ pure @*/ int returnFive(){
11       return 5;
12    }
13 }
```

Fig. 3. Method without a contract

Inlining. KeY and OpenJML have a different approach to handling method calls. OpenJML uses a very puristic approach: any method call will be abstracted by its method specification. Thus, consider the example in Fig. 3. Method `InitPublic` calls method `ReturnFive`. As method `ReturnFive` does not have any method specification, OpenJML will simply assume that any behaviour of this call is possible, and it will not be able to prove the postcondition `a == 5` (even though we can clearly see that the implementation of method `ReturnFive` achieves exactly this).

KeY follows a different approach here. If no postcondition is specified, i.e., no **ensures** clause is present, KeY will inline this method call, and thus the postcondition of method `InitPublic` can be proven. Notice that when a postcondition of `ReturnFive` is specified, even when this is only **ensures true**;, inlining will not happen anymore, and verification of method `InitPublic` will fail (except of course if the postcondition of `ReturnFive` captures that it returns 5).

KeY thus requires a user to think carefully about whether a method call will indeed always end up invoking the same method invocation. If the method `ReturnFive` may be overwritten, the postcondition of method `InitPublic` might not hold anymore after the call to `ReturnFive` in the subclass. Thus, again KeY provides extra flexibility, i.e., not requiring every call to be annotated, but at the risk of verifying something that is not correct. The JML semantics does not explicitly describe whether unfolding is a valid proof step for static verification.

The use of contracts, rather than inlining, can greatly speed up time required for running the automated verification, as witnessed by experiments in KeY done by Knüppel et al. [15]. Obviously, using inlining for program verification can avoid the need of even having to write contracts.

David Cok brought to our attention that OpenJML has some undocumented support for inlining. He suggests the introduction of an inline keyword to steer the desired behavior.

Memory Safety and Exceptional Behaviour. An interesting difference that we noted between KeY and OpenJML is in the checks that are implicitly added to ensure memory safety. The JML semantics advocates a non-null by default semantics, but it does not exclude other exceptions by default.

KeY has three ways of dealing with exceptions, depending on a taclet settings. One setting requires you to prove that no kind of exception will ever be thrown. This does not allow you to verify any program for which throwing errors is part of the specification. Another setting assumes no exceptions occur, making the analysis unsound, but possibly still useful for catching certain kinds of bugs. A final setting, and the one used in the discussion below, is to treat all exceptions as exceptional behavior.

Consider for example the small fragment in Fig. 4. This specification expresses that it will throw an `ArrayIndexOutOfBoundsException`. KeY verifies this example without any problem, but OpenJML does not. Instead, it complains that for the expression `a[-1]` it cannot verify that the index `-1` is within the bounds of the array. Thus: OpenJML adds implicit checks that ensure that this runtime exception will never be thrown, and does not allow the user to prove that this exception actually will be thrown here.

```
1 public class C2 {
2   /*@ private exceptional_behavior
3     @ requires true; // is by default but we have to write something
4     @ signals (ArrayIndexOutOfBoundsException) true;
5     @*/
6   public int getZero(int[] a) {
7           return a[-1];
8   }
9 }
```

Fig. 4. Method with an exception as its contract

As a consequence, implicitly OpenJML reduces the use of exceptional behaviour specifications only to explicit exceptions, and is more rigorous on runtime exceptions than is prescribed by the JML standard [18, Sect. 9.8].

A related difference is that OpenJML checks for bounds within the preconditions of contracts, while KeY does not. Therefore, the example of Fig. 5 results in a warning in OpenJML, while KeY proves its correctness. The JML

```
1 public class C {
2   /*@ private normal_behavior
3     @ requires a[a.length]!=0;
4     @*/
5   public int getZero(int[] a) {
6           return 0;
7   }
8 }
```

Fig. 5. Method raising an exception in its contract

standard states two things about errors in contracts: First, it states that statements in a contract are to be evaluated in order, such that an exception like this can be prevented by verifying that the index is within bounds (i.e. verifying that `a.length<a.length` for our example). Second, it states that a condition is valid if it evaluates to true (also referred to as 'strong semantics'). This means that if exceptions are thrown in the evaluation of a condition, that condition is false. Consequently, this requirement makes the contract trivially valid (see also [3,6] for related discussions). Despite these two statements, we cannot deduce from the JML standard how static verification tools should deal with exceptions in preconditions. We argue that preconditions that evaluate as exceptions are always undesirable and point to errors, agreeing with Chalin [6] on this point. Therefore, we suggest OpenJML's way of dealing with this issue to become standard.

Initialisation Checks. OpenJML and KeY also differ in the checks that they insert for variable initialisation. Consider the example in Fig. 6. JML specifies that variables are always non-null by default, so the `getLength` function satisfies its contract. Therefore, we should reasonably deduce that the `InitArray` function contains an error.

This example is verified in KeY, but OpenJML complains. OpenJML reports that there is no explicit constructor, and mentions line 2 as problematic. Using the non-null default, the reference to array `a` should always be non-null, but this is not guaranteed in this program. Adding the initialisation `a = new int[0];` by uncommenting line 4 solves this issue.

We believe this difference is caused because KeY simply forgets to generate a proof goal for the initialisation of arrays, while OpenJML adds the implicit fact that `a` should be non-null as an implicit class invariant to the variable declaration, and therefore signals a problem.

Power of Underlying Solver. In some cases, the capabilities of the underlying prover determine what can be verified. We previously stated that it can be worthwhile to combine different provers, depending on which prover is most suitable for the task. In doing this, we implicitly assume that differences exist in which annotated programs can be proven automatically, and which cannot. We show that this is indeed the case, even though this does not give us fundamental

```
1 public class InitArray {
2   private int[] a;
3   InitArray(){
4     // a = new int[0]; // missing
5   }
6   /*@ ensures \result >= 0; */
7   public int getLength() {
8     return a.length;
9   }
10 }
```

Fig. 6. No array initialisation

insights into how KeY and OpenJML interpret programs. We give three examples: one where both provers fail, a second where OpenJML is able to prove correctness automatically while KeY is not, and a third example where it is the other way around.

Consider first the code fragment in Fig. 7.

```
1 public class ReverseArray_failing {
2  /*@ public normal_behavior  diverges true; @*/
3  public void reverse(int[] a) {
4   int i = 0;
5   final int length = (a.length/2) ;
6   /*@
7     @ loop_invariant (\forall int j; j>=0 && j<i;
8                        \old(a[a.length-(j+1)])==a[j])
9     @  && (\forall int j; j>=i && j<length;
10    @          \old(a[a.length-(j+1)])==a[a.length-(j+1)] &&
11    @          \old(a[j])==a[j]);
12    @ loop_invariant i>=0 && i<=length;
13    @*/
14   while (i<length) {
15     int tmp = a[a.length-(i+1)];
16     a[a.length-(i+1)] = a[i];
17     a[i] = tmp;
18     i++;
19   }
20  }
21 }
```

Fig. 7. Complicated loop invariant

This program, with the loop invariant as specified, could not be verified automatically by KeY: After letting KeY run automatically for an hour, still no solution was found. We believe a manual KeY proof exists, as this is claimed for a more elaborate version of this code [14].

When we try to verify this with OpenJML, verification fails within ten seconds. This makes the example one where both OpenJML and KeY fail to verify a program. Fortunately there was an easy fix, by splitting the big loop invariant into two separate loop invariants. That is: replace the `&&` on line 9 by `; loop_invariant`. This verified the program without any problem, again in roughly ten seconds. Notice that this is logically completely equivalent, but apparently using the full conjunction in the generated proof obligation is too complicated for the underlying first-order prover Z3. Thus, the OpenJML user has to be aware of this issue, and make sure that his or her specification style fits the capabilities of the underlying prover. If we run KeY on the changed program, it again fails to find a solution (in reasonable time).

For our second example, we did not manage to solve the issue. Consider the Least Common Prefix (LCP) program in Fig. 8, which is part of a solution to a VerifyThis 2012 challenge [4,11]. Verification of this algorithm works in KeY, but not OpenJML. David Cok pointed out that OpenJML can verify the example by replacing the maintaining clause on line 13 and 14 with:

```
(\forall int z; x <=z && z < x+l; a[z] == a[y+z-x])
```

David Cok also points out that the issue is indeed with the underlying Z3 solver, which has trouble with quantified expressions that have arbitrary expressions as array indices.

```
1  /* @author bruns, woj */
2  final class LCP {
3
4    /*@ normal_behavior
5    @ requires 0 <= x && x < a.length;
6    @ requires 0 <= y && y < a.length;
7    @ requires x != y;
8    @ pure @*/
9    static int lcp(int[] a, int x, int y) {
10     int l = 0;
11     /*@ maintaining 0 <= l && l+x <= a.length
12       @             && l+y <= a.length && x!=y;
13       @ maintaining (\forall int z; 0 <= z && z < l;
14       @                         a[x+z] == a[y+z] );
15       @ decreasing a.length-l; @*/
16     while (x + l < a.length && y + l < a.length
17          && a[x + l] == a[y + l])
18       l++;
19     return l;
20   }
21 }
```

Fig. 8. Modified LCP example

5 Lessons Learned

This case study investigated differences among JML, OpenJML and KeY. Both tools aim to verify JML-annotated Java programs, but this case study shows that the differences in their behaviour are substantial. Therefore, at the moment it is a non-trivial exercise to reuse verified specifications from one tool by the other tool, even though the developers of OpenJML and KeY have had discussions to agree on a common semantics for a core of JML.

The differences fall in different categories: syntax of the specification language, interpretation of the JML semantics, behaviour of the underlying prover, and choice of defaults in programs and specifications. To improve interoperability between tools, we need to investigate if we can reduce these differences and if this is not possible, we should make sure that we document them. We believe that tool developers should take much more responsibility than they do currently to improve interoperability between tools. Looking at the different categories of differences that we identified, we believe the following should be aimed for:

- Syntactical differences should simply be avoided. If tool developers feel the need to define their own syntax, we believe that they should provide users with a script to turn the annotated program into a standard-JML compliant version, or use special markers for the non-JML-compliant annotations.
- Differences in behaviour caused by the underlying prover should be avoided as much as possible. These are caused by the format in which proof obligations are sent to the underlying prover (and by the use of different underlying provers). Finding the optimal format is a research challenge, and it is important that tool developers exchange their experiences with this. The issue can probably also be further reduced by supporting different back-end provers.
- The differences due to a different interpretation of the JML semantics should be avoided as much as possible. Therefore, it is important to continue the discussion on a common semantics of core JML, and to document the outcome of this discussion. Also, tool developers should agree to adhere to the decisions made during this discussion, and if necessary, adapt their tool implementation. If a tool developer still decides to deviate from this common semantics, he or she should document this, or preferably provide a flag that allows one to still use the common semantics.
- For the differences caused by the choice of defaults in programs and specifications, the same applies: these should be documented, and an option could be provided as a special flag. In some cases, the tool might also decide to issue an explicit warning about the defaults chosen, and that other tools might deviate from this. For example, it would help if the KeY tool would issue a warning that it did not check whether the Java program actually can be compiled.

And very importantly, these choices and assumptions that cause differences should be documented in a way that is understandable and accessible for people who did not develop KeY or OpenJML, as they are ones that are the most likely to benefit from tool interoperability. Ideally, tools should be developed with this

idea of interoperability in mind. We understand that it might not be easy to change the complete implementation of a tool, but it would help users a lot if OpenJML could be invoked with a `-KeY` flag, and vice versa[1].

To improve the current situation, a first starting point would be to define a collection of verification benchmarks with intended behaviours (similar to the litmus tests for relaxed memory models). The examples discussed in this paper could be a starting point for this, but further extensions will be necessary.

In this case study, and also in the conclusions, we focused very much on JML-annotated programs. However, we believe that the general lessons learned also apply to other verification tools, and that it is time for the formal verification community to really put more effort in tool interoperability, in order to increase the impact of formal verification.

Acknowledgements. The work of Huisman and Joosten is supported by NWO grant 639.023.710 for the Mercedes project.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification - The KeY Book, LNCS, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Beckert, B., Klebanov, V., Weiß, B.: Dynamic logic for Java. Deductive Software Verification – The KeY Book. LNCS, vol. 10001, pp. 49–106. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6_3
3. Drossopoulou, S., Eisenbach, S., Jacobs, B., Leavens, G.T., Müller, P., Poetzsch-Heffter, A.: Formal techniques for Java programs. In: Goos, G., Hartmanis, J., van Leeuwen, J., Malenfant, J., Moisan, S., Moreira, A. (eds.) ECOOP 2000. LNCS, vol. 1964, pp. 41–54. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44555-2_4
4. Bruns, D., Mostowski, W., Ulbrich, M.: Implementation-level verification of algorithms with KeY. STTT **17**(6), 729–744 (2015)
5. Burdy, L., et al.: An overview of JML tools and applications. STTT **7**(3), 212–232 (2005)
6. Chalin, P.: Reassessing JML's logical foundation. In: Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP 2005), Glasgow, Scotland (2005)
7. Cheon, Y.: A Runtime assertion checker for the java modeling language. Ph.D. thesis, Department of Computer Science, Iowa State University, Ames (2003). Technical Report 03–09
8. Cok, D.R., Kiniry, J.R.: ESC/Java2: uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30569-9_6

[1] David Cok tells us that OpenJML can be invoked with a `-strict` flag, causing it to follow the JML semantics more strictly, which is already a great step.

9. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 472–479. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_35
10. Griffioen, D., Huisman, M.: A comparison of PVS and Isabelle/HOL. In: Grundy, J., Newey, M. (eds.) TPHOLs 1998. LNCS, vol. 1479, pp. 123–142. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055133
11. Huisman, M., Klebanov, V., Monahan, R.: VerifyThis 2012 - a program verification competition. STTT **17**(6), 647–657 (2015). https://doi.org/10.1007/s10009-015-0396-8
12. JML support in KeY. https://www.key-project.org/jml-support-in-key/
13. Currently supported features, April 2018. http://www.openjml.org/documentation/features.shtml
14. Kirsten, M.: Proving well-definedness of JML specifications with KeY. Master's thesis, ITI Schmitt, Karlsruhe Institute of Technology, November 2013
15. Knüppel, A., Thüm, T., Padylla, C., Schaefer, I.: Scalability of deductive verification depends on method call treatment. In: 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA). LNCS, Springer, Heidelberg (2018, to appear)
16. Leavens, G.T., Baker, A.L., Ruby, C.: JML: a notation for detailed design. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems. SECS, vol. 523, pp. 175–188. Springer, Boston (1999). https://doi.org/10.1007/978-1-4615-5229-1_12
17. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. ACM SIGSOFT Softw. Eng. Notes **31**(3), 1–38 (2006)
18. Leavens, G., et al.: JML Reference Manual, Department of Computer Science, Iowa State University, February 2007. http://www.jmlspecs.org
19. Rustan, K., Leino, M.: Applications of extended static checking. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 185–193. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-47764-0_11
20. 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). https://doi.org/10.1007/978-3-540-78800-3_24
21. Thüm, T., Meinicke, J., Benduhn, F., Hentschel, M., von Rhein, A., Saake, G.: Potential synergies of theorem proving and model checking for software product lines. In: Proceedings of the 18th International Software Product Line Conference, SPLC 2014, vol. 1, pp. 177–186. ACM (2014)
22. Weiß, B.: Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction. Ph.D. thesis, Karlsruhe Institute of Technology (2011)

Design and Verification of Restart-Robust Industrial Control Software

Dimitri Bohlender(✉) and Stefan Kowalewski

Embedded Software, RWTH Aachen University, Aachen, Germany
{bohlender,kowalewski}@embedded.rwth-aachen.de

Abstract. Many systems in automated production and industrial automation operate in safety-critical environments and must meet rigorous safety requirements. To enable safe operation even in the case of a power outage, the PLCs driving these systems feature battery-backed memory areas to prevent loss of data and allow for implementation of resumption strategies. However it is up to an automation engineer to decide which variables to retain, and errors that only occur after program restart are a common problem in industrial control code.

We present approaches to both verifying the absence of such errors and synthesising safe configurations of retain variables with off-the-shelf tooling. The synthesis problem reduces to solving particular exists-forall quantified Horn clauses, for what we also propose a more efficient counterexample-guided procedure.

Evaluation of our prototypical implementation on examples from the PLCopen Safety library shows the techniques' strengths and limitations.

Keywords: Software verification · Parameter synthesis
Restart-robustness · Integration of formal methods
Programmable logic controllers

1 Introduction

In industrial applications, such as chemical plants or assembly lines, control software must meet high safety and reliability requirements as errors may entail significant costs and hazards. *Programmable logic controllers* (PLCs) are rugged computers which are particularly tailored to, and widely used in, the industrial automation domain.

The IEC 61131 standard defines requirements to both hardware and software aspects of PLCs as well as their cyclic mode of operation, i.e. reading inputs (from sensors), executing a main program, writing outputs (to actuators) and starting all over. To enable the design of systems that we call *restart-robust* w.r.t. some specification, i.e. whose behaviour complies with the specification even when resuming operation after a restart, the IEC 61131-3 defines a *retain* qualifier for variables in PLC programming languages. Variables that are declared this way, are stored in a dedicated battery-backed memory s.t. their values are available even after a power outage.

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 47–68, 2018.
https://doi.org/10.1007/978-3-319-98938-9_4

For example consider the use case of automated drilling of holes in workpieces. If the drill's position or mode of operation are not retained, a restart may result in unintended movement of the drill and damage to the system itself, the payload, or persons within reach – even if such a malfunction were not possible in a restart-free operation.

While retain variables are standardised, the semantics of an assignment to such a variable is not, and left to the PLC vendors. In this work, we focus on the two most prominent implementations supported by major development environments for PLC software: *immediate* and *delayed* writing of retain variables. In the former case, an assignment to a retain variable is translated to an immediate write to the battery-backed memory. However, frequent writing to this memory is often slower than accessing the main memory, and every immediate write increases the number of corner-cases to consider when developing restart-robust applications. By way of contrast, in the case of delayed writing, assignments to retain variables during program execution are in fact writes to the main memory. The actual copying of these values to the battery-backed memory is delayed until the end of the current PLC cycle. Depending on the application or PLC vendor, one or the other semantics may have to be supported.

Due to the cyclic operation of PLCs, where outputs are only written at the end of a program execution, the intermediate states of a PLC are not visible to the environment. Therefore, when automation engineers or specifications talk about a PLC's state they implicitly refer to its *observable state*. Although most specifications in this domain are formulated in natural language, they can usually be expressed formally in terms of invariants or temporal logics [16]. When developing restart-robust control software or upgrading existing functionality to handle restarts safely, it becomes an automation engineer's task to manually determine which variables must be retained without violating a given specification of safe behaviour, and implement the functionality needed for resuming operation. Since mistakes can easily be made, but be very subtle and hard to detect, this is a common problem in industrial control code [20].

Contribution. The primary contribution of this paper is the design of automated verification procedures that aid in the engineering of restart-robust logic control software. To this end,

1. we formalise the restart behaviour for delayed and immediate write semantics, and sketch its integration with established approaches for PLC software verification,
2. we show how these characterisations can be extended, to acquire procedures that synthesise restart-robust configurations of retain variables using off-the-shelf tools,
3. we propose a dedicated counterexample-guided procedure, which exploits specifics of the problem, and makes synthesis of configurations practical in the first place,

4. we evaluate our approaches to both verification of a program's restart-robustness and synthesis of safe configurations of retain variables using examples from the *PLCopen Safety* [31,32] library, and
5. we provide all the artefacts needed to reproduce our results, or even improve upon.

Related Work. Due to the safety critical nature of industrial automation, the use of formal verification is advisable and many successful applications of formal methods have been reported in the past. However most work operates on model level, analysing drafts and models of the system to be implemented, instead of the actual implementation [30]. While such analyses are necessary to find conceptual problems early in the development cycle, they do not guarantee that the implementation will be free of bugs.

The endeavour of verifying PLC software goes back to Moon [28], who used the SMV formalism [27] to characterise programs written in the *Ladder Diagram* programming language. Although SMV targets hardware verification, and Ladder Diagram indeed is a circuit-like language without control flow, most present day PLC software verifiers still use SMV-based tooling for model checking higher-level PLC programming languages [3,16,30]. However, with *constrained Horn clauses* (CHCs) increasingly becoming a basis for automatic program verification in recent years [7], they have been adopted in verification of logic control software too [9,10]. Therefore, we examine the characterisation of restart semantics in both formalisms.

To the best of our knowledge, we are the first to investigate formal verification of a program's restart-robustness and synthesis of safe retain configurations. The only directly related work [20] assumes delayed write semantics and adapts static value analysis to distinguish between variables' values before and after a restart. *Crash recoverability* of C programs [24] is a related problem, using a similar modelling, but differing from restart-robustness in terms of requirements and program transformations.

The search for constants s.t. a system satisfies some property is commonly referred to as *parameter synthesis*, and we model the search for safe retain configurations as such. Besides of our characterisation of the problem in terms of the SMV formalism, SMV-based tooling has also been used in bioinformatics to find parameters for models of gene regulatory networks [2]. Our counterexample-guided approach is most similar to [13] but does not require quantifier elimination, is independent of the chosen theory to model values, and works with any CHC-solving algorithm.

Outline. We commence with an example program, illustrating the concrete problems and expected solutions. Section 3 recapitulates the formal concepts relevant for characterisation of these problems in terms of existing formalisms (Sect. 4) and understanding of the counterexample-guided synthesis procedure (Sect. 5). In Sect. 6, we present experimental results and provide concluding remarks in Sect. 7.

2 Motivating Example

Consider the program from Fig. 1, which picks up on the example used in [20], but is slightly modified to make a different point. For simplicity, it does not feature input variables and operates on two integer variables `a` and `b`, and a retentive Boolean flag `fs`. Intuitively, the flag `fs` is used to track whether the program is in its first cycle, s.t. the initialisation of `b` (cf. line 11) is only executed once. The program starts with the explicitly provided default initialisation $[fs \mapsto true, a \mapsto 0, b \mapsto 0]$.

```
 1  PROGRAM Program1
 2    VAR RETAIN
 3      fs:BOOL := TRUE;
 4    END_VAR
 5    VAR
 6      a:INT := 0;
 7      b:INT := 0;
 8    END_VAR
 9    IF fs THEN
10      fs := FALSE;
11      b := 2;
12    END_IF
13    a := 1234/b;
14  END_PROGRAM
```

Fig. 1. Running example program

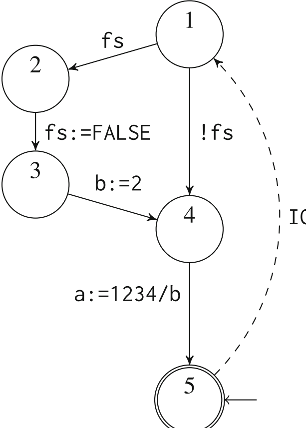


Fig. 2. CFA of the running example

Let $a \geq 0$ be the invariant that needs to hold for every observable state. In a regular execution this is indeed true. Both `a` and `b` are initially set to 0, but `b` is set to 2 during the first cycle which results in `a` being set to $1234/2 = 617$. Since `fs` is only *true* in the first cycle, the values of `b` and `a` stay like that forever.

However, the program is not restart-robust w.r.t. to $a \geq 0$ in the context of delayed write semantics for retain variables. If a restart occurs after the first cycle, i.e. once $\neg fs \wedge b = 2$ holds, `fs` will stay unchanged but `b` will be reset to 0 and let `a` take an arbitrary value, by causing an undefined division by 0. Since immediate write semantics allow for a superset of the behaviours of delayed write semantics, the program is not restart-robust for those either.

The next question is, whether it is possible to fix the program by changing which variables to retain. The variable `fs` that is being retained currently, is assumed to be retained for a reason and not in question to become volatile, but close inspection and intuition will help with identifying `b` as a suitable candidate. If `b` is retained too, the program becomes restart-robust w.r.t. $a \geq 0$, since the divisor in line 13 is always 2.

Nevertheless, if the program is used in the context of immediate write semantics for the retain variables, it suddenly becomes not restart-robust again. Although the program is simple, the violating run is easy to miss and will eventually lead to unexpected behaviour. With immediate write semantics, a restart might occur after the write to `fs` but before setting `b` to 2, i.e. leaving it at its initial 0. Since $\neg fs$ will hold, there will be no initialisation and the division by 0 will be reached again. In fact, in this case there is no way to achieve restart-robustness by changing the configuration of retain variables.

Keeping track of all the possible (mis-)behaviour in the context of restarts is clearly prone to human error. It should not be surprising that unexpected behaviour after a restart is a common problem [20], given that implementation of restart-robust systems is currently approached without aid of automatic procedures.

3 Preliminaries

3.1 Program Representation

We restrict the presentation to a reduced programming language, featuring only *assignments*, *assumes* and a *havoc* instruction, which models the assignment of a nondeterministic value, visualised as $x := e$, g and $x :=?$ respectively, where x is a variable, e an expression, and g a Boolean expression acting as a guard. This is a common approach [1,7,8], coming without loss of generality. In particular, all calls can be inlined since recursion is prohibited in PLC programs.

We use *Instr* to denote the set of such instructions, and represent a program using the notion of a *control flow automaton* (CFA).

Definition 1 (Control Flow Automaton). *A CFA $\mathcal{A} = (L, E)$ is a directed graph, where the vertices L are the program locations, and the edges $E \subseteq L \times Instr \times L$ model the program's instructions and their effect on control flow.*

Definition 2 (Program). *A program $P = (\mathbf{X}, \mathbf{X}_{in}, \mathcal{A}, l_{EoC}, l_{init}, def)$ consists of a set of variables $\mathbf{X}$, input variables $\mathbf{X}_{in} \subseteq \mathbf{X}$, a CFA $\mathcal{A}$ whose instructions refer to the variables from $\mathbf{X}$, the end-of-cycle location $l_{EoC} \in L$, the initial program location $l_{init} \in L$, and a partial mapping def from variables to their default values. The characterisation $I(\mathbf{X})$ of initial values is implicitly given by the defined defaults, i.e. $\bigwedge_{x \in \mathbf{X}} x = def(x)$.*

Note that l_{EoC} is both the only location where a PLC's state is observable and also the initial program location, although the formalism allows for them to differ, e.g. as a result of instrumentation (cf. Sect. 4.1). Similarly, while all variables of a logic control application have well-defined default values (cf. IEC 61131), later modelling steps may introduce variables for which *def* is undefined and the initial value nondeterministic.

Figure 2 illustrates the CFA that models our running example. For the sake of readability and intuition, we use an `IO` instruction to model the PLC's reading

of sensor values at the beginning of a new cycle, which is syntactic sugar for a sequence of $x :=?$ for each $x \in \mathbf{X}_{\mathrm{in}}$. Since this particular program has no input variables anyway, it can be thought of as a plain *goto*, representable as the assume $(5, \mathit{true}, 1) \in E$.

Semantics. The *state* of a program is an assignment σ that maps each variable from $\{pc\} \cup \mathbf{X}$ to a value, where pc represents the program counter, i.e. $\sigma(pc) \in L$.

Since a CFA is essentially a GOTO-program, its transition relation $T \subseteq \Sigma \times \Sigma$, where Σ denotes the set of all states, can be derived from the *weakest-preconditions* of unstructured programs [1], i.e.

$$T(pc, \mathbf{X}, pc', \mathbf{X}') = \bigwedge_{(l,instr,l') \in E} (pc = l \rightarrow [\![instr]\!] \wedge pc' = l'), \tag{1}$$

where the primed variables' instances $\{pc'\} \cup \mathbf{X}'$ denote the next-state valuation and $[\![\cdot]\!]$ is the instruction's characterisation:

$$[\![instr]\!] = \begin{cases} (\bigwedge_{v \in \mathbf{X} \setminus \{x\}} v' = v) \wedge x' = e & instr = (x := e) \\ (\bigwedge_{v \in \mathbf{X} \setminus \{x\}} v' = v) & instr = (x :=?) \\ (\bigwedge_{v \in \mathbf{X}} v' = v) \wedge g & instr = g \end{cases} \tag{2}$$

3.2 Symbolic Model Verifier

The *Symbolic Model Verifier* (SMV) formalism allows the symbolic definition of a transition system $S = (\mathbf{V}, I, T)$ in terms of a characterisation of the initial states I over the variables $\mathbf{V}$, and a transition relation T as seen in the previous section. Accordingly, modelling the program semantics for SMV-based verifiers is a straight-forward reuse of Eq. (1). Note though that SMV targets hardware-verification, and with $\mathbf{V} = \{pc\} \cup \mathbf{X}$ the program counter is treated like any other variable. Therefore, if the control flow is to be exploited by a SMV-based verifier, some variant of *large-block encoding* [4] has to be employed.

The SMV formalism allows the definition of specifications in terms of *invariants* and *temporal logics* – in particular CTL [14]. However, one needs to exercise caution when expressing a specification for PLC software in SMV, since SMV specifications are interpreted in the step-size that T is provided in, e.g. a single instruction per step, while the original specification only refers to the observable states, i.e. a step is a whole execution cycle. For an invariant $\varphi(\mathbf{X})$ this can be easily accounted for by only checking it at the end-of-cycle location, i.e. use $pc = l_{\mathrm{EoC}} \rightarrow \varphi(\mathbf{X})$. Although we focus on invariants, the need for reformulation of specifications can generally be avoided by characterising the whole program as a single step [3,8].

3.3 Constrained Horn Clauses in Software Verification

Definition 3 (Constrained Horn Clause). *Given sets of variables $\mathcal{V}$, function symbols $\mathcal{F}$, and predicates $\mathcal{P}$, a* constrained Horn clause *(CHC) is a formula*

$$\forall \mathcal{V}\ \underbrace{p_1(\mathbf{X}_1) \wedge \cdots \wedge p_k(\mathbf{X}_k) \wedge \varphi}_{body} \rightarrow h(\mathbf{X}),\ \ k \geq 0,$$

where φ is a constraint over $\mathcal{F}$ and $\mathcal{V}$, $\mathbf{X}_i, \mathbf{X} \subseteq \mathcal{V}$ are possibly empty vectors of variables, and $p_i(\mathbf{X}_i)$ is an application of a predicate p_i of arity $|\mathbf{X}_i|$.

We use *body* to refer to the antecedent of the CHC and *head* to denote h. A CHC is called a *query* if its head is free of $\mathcal{P}$ symbols and otherwise, it is called a *rule*. Following the convention of logic programming literature, we use the shorthand notation

$$h(\mathbf{X}) \leftarrow p_1(\mathbf{X}_1), \ldots, p_k(\mathbf{X}_k), \varphi. \tag{3}$$

A set of CHCs is satisfiable if there exists an interpretation of the predicates that satisfies each φ. As illustrated by [22], intuitively, each p_i represents an unknown over-approximate summary, while a query defines a property to be proved. In the context of CFAs, the p_i correspond to over-approximations of the reachable valuations at program location i. Therefore, checking whether a program satisfies a safety property, amounts to establishing the satisfiability of CHCs that encode the corresponding verification conditions, as shown below.

Following [7], a program $P = (\mathbf{X}, \mathbf{X}_{\text{in}}, (L, E), l_{\text{EoC}}, l_{\text{init}}, \mathit{def})$ is characterised by

$$p_{\text{init}}(\mathbf{X}) \leftarrow I(\mathbf{X}) \tag{4}$$

$$p_{l'}(\mathbf{X}') \leftarrow p_l(\mathbf{X}), [\![\mathit{instr}]\!] \text{ for each } (l, \mathit{instr}, l') \in E \tag{5}$$

Note that in contrast to the p_i, I is not uninterpreted but explicitly given (cf. Sect. 3.1).

To prove that the program complies with an invariant φ, we check whether an interpretation of predicates p_i exists s.t. all CHCs are satisfied and the over-approximation of observable states $p_{\text{EoC}}(\mathbf{X})$ subsumes the safe states (cf. [26]), by adding the query

$$\varphi(\mathbf{X}) \leftarrow p_{\text{EoC}}(\mathbf{X})\,. \tag{6}$$

4 Modelling the Restart Semantics

Existing approaches for PLC software verification formalise only the *nominal* program semantics, implementing the approaches from Sect. 3, and ignoring possible restarts. In the following, we illustrate how restarts with delayed and immediate write semantics for retain variables can be modelled in terms of these established formalisms, to allow reuse of existing verification machinery.

Similar to an interrupt, a restart may occur at any time during program execution. When that happens, the program counter is reset to l_{EoC}, and the next execution cycle starts with all non-retain variables reinitialised with their default values. The retain variables, however, take their corresponding values that are stored in the battery-backed memory at that time. Note that marking a variable as retained does not imply that all assignments to it are immediately reflected in the battery-backed memory – this depends on the employed retain semantics.

Keep in mind that for PLC programs, the initial and end-of-cycle location are identical, and the following instrumentations are to be employed prior to other modifications that may introduce a distinct entry l_{init} for modelling purposes (cf. Definition 7).

Delayed Write Semantics. If a logic controller is restarted in the middle of an execution cycle and writing to retain variables is realised via delayed write semantics, there will not have been any write to the battery-backed memory since the end of the previous cycle. The resulting state will have all non-retain variables reset to their initial values, and the retain variables back at the values they had at the end-of-cycle location.

Since a cycle's nominal semantics becomes irrelevant if a restart happens during its execution, we model such a restart by a nondeterministic choice at the end-of-cycle location. If a restart is chosen to occur, we can keep the current values of all retain variables and reinitialise the others, otherwise we execute the program's nominal semantics.

Note that power outages during the delayed writes are omitted in the modelling since these writes can be handled atomically by the PLC's operating system, e.g. by using auxiliary memory-backed variables that are written immediately.

Definition 4 (Delayed Write Instrumentation). *Given a set of retain variables* $\mathbf{X}_{ret} \subseteq \mathbf{X}$ *for a program* $P = (\mathbf{X}, \mathbf{X}_{in}, (L, E), l_{EoC}, l_{EoC}, def)$*, its* delayed write instrumentation *yields a program*

$$P_{dw} = (\mathbf{X}, \mathbf{X}_{in}, (L \uplus L_{init}, E \uplus E_{init} \uplus E_{restart}), l_{EoC}, l_{EoC}, def)$$

where

- $L_{init} := \{l_x \mid x \in \mathbf{X} \setminus \mathbf{X}_{ret}\}$ *are new program locations in between which the resetting of values occurs – one for each non-retain variable,*
- $E_{init} := \{(l_{x_1}, x_1 := def(x_1), l_{x_2}), \ldots, (l_{x_n}, x_n := def(x_n), l_{EoC})\}$ *are the reinitialising assignments for every non-retain variable,*
- $E_{restart} := \{(l_{EoC}, true, l_{x_1})\}$ *models a restart during the execution of the cycle,*

with the non-retain variables denoted by $x_1, \ldots, x_n = \mathbf{X} \setminus \mathbf{X}_{ret}$.

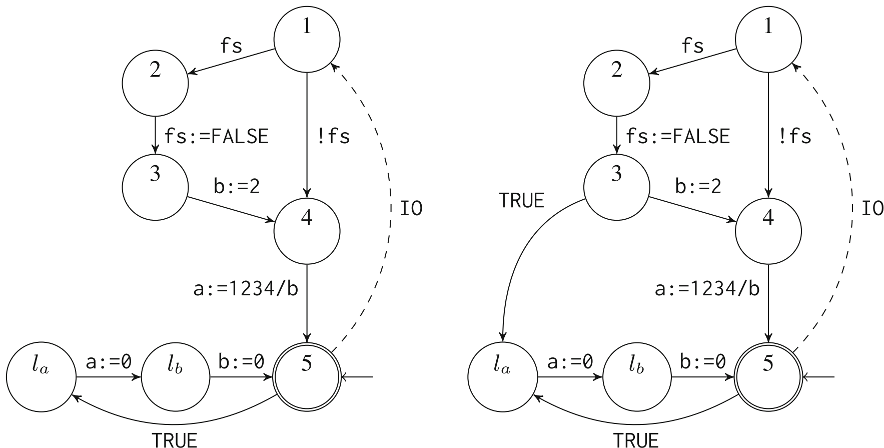


Fig. 3. Delayed write instrumentation

Fig. 4. Immediate write instrumentation

Figure 3 illustrates the result of applying this instrumentation to our running example. At the beginning of every execution cycle, either the edge $(5, \mathit{true}, l_a)$ leading to the reinitialisation, or the `IO`-edge leading to the nominal cycle semantics will be taken.

Immediate Write Semantics. If a logic controller is restarted in the middle of an execution cycle and writing to retain variables is realised via immediate write semantics, all the assignments to retain variables on the path from the end-of-cycle location to the location where the restart occurred will be reflected in the battery-backed memory. The resulting state will have all non-retain variables reset to their initial values, and the retain variables at the values they had at the time of the restart.

Since only assignments to retain variables change the resulting state after a restart, it suffices to model restarts with a nondeterministic choice after every write to a retain variable – instead of a choice in every location. Note that, although a restart that occurs before any write to a retain variable does not lead to a new state, and is irrelevant for checking invariants, it may still cause violations of other temporal specifications. To model the restarting before any write to a retain variable, we also add a nondeterministic choice at the end-of-cycle location, as in the case of delayed write semantics.

Definition 5 (Immediate Write Instrumentation). *Given a set of retain variables* $\mathbf{X}_{ret} \subseteq \mathbf{X}$ *for a program* $P = (\mathbf{X}, \mathbf{X}_{in}, (L, E), l_{EoC}, l_{EoC}, \mathit{def})$*, its* immediate write instrumentation *yields a program*

$$P_{iw} = (\mathbf{X}, \mathbf{X}_{in}, (L \uplus L_{init}, E \uplus E_{init} \uplus E_{restart}), l_{EoC}, l_{EoC}, \mathit{def})$$

where

- $L_{init} := \{l_x \mid x \in \mathbf{X} \setminus \mathbf{X}_{ret}\}$ *are new program locations in between which the resetting of values occurs – one for each non-retain variable,*
- $E_{init} := \{(l_{x_1}, x_1 := def(x_1), l_{x_2}), \ldots, (l_{x_n}, x_n := def(x_n), l_{EoC})\}$ *are the reinitialising assignments for every non-retain variable,*
- $E_{restart} := \{(l_{EoC}, true, l_{x_1})\} \cup \{(l', true, l_{x_1}) \mid (l, x := e, l') \in E, x \in \mathbf{X}_{ret}\}$ *models a restart before any and after every write to a retain variable during the cycle,*

with the non-retain variables denoted by $x_1, \ldots, x_n = \mathbf{X} \setminus \mathbf{X}_{ret}$.

Note that with L_{init} and E_{init} being the same as in the delayed write instrumentation, but more cases of restarts to consider, the immediate write instrumentation yields a superset of its behaviours. As a result, if a program is restart-robust in the context of these semantics, it is also restart-robust in the context of delayed write semantics. If a program is not restart-robust in the context of delayed write semantics, it will not be in the context of these semantics either.

Figure 4 illustrates the result of applying this instrumentation to our running example. Besides the restart edge at the end-of-cycle location, which models a restart occurring prior to any write to the battery-backed memory, we now also consider a restart after the write to *fs*, since it was declared as a retain variable.

To check whether a program is restart-robust w.r.t. to some specification, we can now use the appropriate instrumentation to reduce the problem to something, that we already have verification procedures for (cf. Sect. 3).

4.1 Characterising Parameter Synthesis with CHCs and SMV

While the proposed reductions enable checking a program's restart-robustness w.r.t. some specification, they do not aid the developing engineer in actually designing programs that are restart-robust, or upgrading existing modules to enable restarts-robustness by choosing appropriate retain variables. Therefore, this section examines how the presented reductions can be modified, s.t. existing tooling can also be used to synthesise configurations of retain variables that make the program restart-robust w.r.t. a property of interest.

To enable the examination of different configurations of retain variables, the configuration itself must become a *parameter* of the model. To this end, we add Boolean constants to the model, one for each non-retain variable, which encode whether the corresponding variable is to be retained. The constants' values are nondeterministically chosen at the start of the program and used to parametrise the reinitialisation semantics. Furthermore, they are used to guard the restarts that depend on whether a particular variable is retained, e.g. in the case of immediate write semantics.

Definition 6 (Parametrisation of Retains). *Given the result of a delayed or immediate write instrumentation* $P = (\mathbf{X}, \mathbf{X}_{in}, (L \uplus L_{init}, E \uplus E_{init} \uplus E_{restart}), l_{EoC}, l_{EoC}, def)$ *and the used retain variables* $\mathbf{X}_{ret} \subseteq \mathbf{X}$*, its* parametrisation of retains *yields a program*

$$P_{par} = (\mathbf{X} \uplus \mathbf{X}_{par}, \mathbf{X}_{in}, (L \uplus L_{init}, E \uplus E_{parInit} \uplus E_{restart} \uplus E_{parRestart}), l_{EoC}, l_{EoC}, def)$$

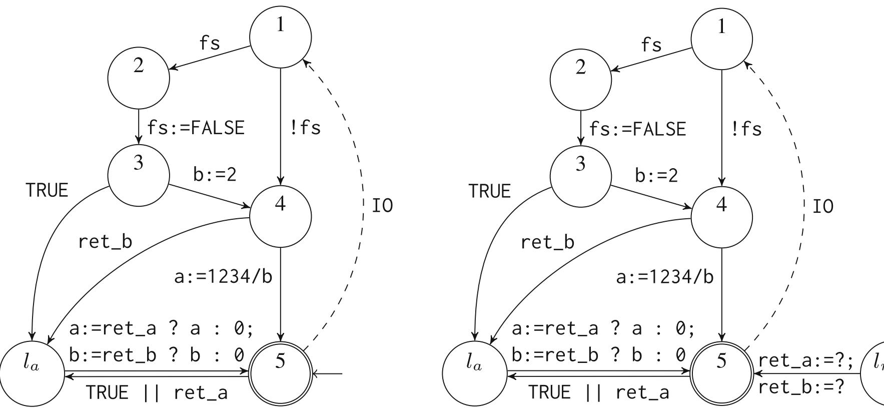


Fig. 5. Immediate write instrumentation with dependence on retain configuration

Fig. 6. SMV-based synthesis requires choice of retain variables to be part of the model

where the non-retain variables are still denoted by $x_1, \ldots, x_n = \mathbf{X} \setminus \mathbf{X}_{ret}$*, and*

- $\mathbf{X}_{par} := \{ret_x \mid x \in \mathbf{X} \setminus \mathbf{X}_{ret}\}$ *are new Boolean variables that parametrise which of the currently non-retained variables to treat as retained,*
- $E_{parInit} := \left\{ \begin{array}{c} (l_{x_1}, x_1 := ret_{x_1}?x_1 : def(x_1), l_{x_2}), \ldots, \\ (l_{x_n}, x_n := ret_{x_n}?x_n : def(x_n), l_{EoC}) \end{array} \right\}$
are the parametrised reinitialising assignments, using ternary if expressions,
- $E_{parRestart} := \begin{cases} \emptyset & \textit{delayed write} \\ \{(l', ret_x, l_{x_1}) \mid (l, x := e, l') \in E, x \in \mathbf{X} \setminus \mathbf{X}_{ret}\} & \textit{otherwise} \end{cases}$
models a restart after every write to a variable that can be parametrised to be retained in the case of immediate write semantics.

Figure 5 illustrates the result of parametrising the retain configuration, applied to an immediate write instrumentation of our example program. In comparison to Fig. 4, every static reinitialisation of a non-retain variable x has been replaced with an expression dependent on a parameter ret_x, and additional guarded edges that lead to the resetting have been added after assignments to potential retain variables. Note that for the sake of readability, the visualisation of the reinitialisation sequence is simplified in that both assignments are presented as a sequence on a single edge, instead of featuring the intermediate location l_b as in previous figures.

However, analysing a parametrised program with the techniques from Sect. 3 will not result in checking whether a restart-robust retain configuration exists, but whether all possible retain configurations (of not yet retained variables) are restart-robust.

Existential Quantification. To determine whether a retain configuration exists that makes the program restart-robust w.r.t. a property of interest, we illustrate how the introduced parameters can be existentially quantified in the context of both the CHC and SMV formalism.

Since the variables in CHCs are implicitly universally quantified, the synthesis problem requires us to move on to the more complex case of *exist-forall* quantified CHCs. Expressing parameter synthesis in this class of Horn clauses is straightforward. We keep the actual clauses as illustrated in Sect. 3.3, but replace the quantification $\forall \mathcal{V}$ by $\exists \mathbf{X}_{\text{par}} \forall \mathcal{V} \setminus \mathbf{X}_{\text{par}}$. The resulting constrains are satisfiable if interpretations for both the parameters $x \in \mathbf{X}_{\text{par}}$ and utilised predicates $p \in \mathcal{P}$ exist, s.t. the clauses are satisfied for all values of the remaining variables $\mathcal{V} \setminus \mathbf{X}_{\text{par}}$. The downside is that a solver will not be able to use its efficient procedures tailored to solving universally quantified Horn clauses, but resort to general techniques for *satisfiability modulo theories* (SMT) [25].

While the SMV formalism itself does not support quantification, a CTL specification may reason about the existence of a path. In combination with another modification of the CFA, the existence of a certain retain configuration can be reduced to the existence of a path. Intuitively, we prepend a nondeterministic choice of a retain configuration to the original program entry l_{EoC}, and question the existence of a path through this choice s.t. from l_{EoC} on the program exhibits restart-robust behaviour.

Definition 7 (Integration of Parameter Choice). *Given a delayed or immediate write instrumented and parametrised program* $P = (\mathbf{X} \uplus \mathbf{X}_{par}, \mathbf{X}_{in},)$ $(L, E), l_{EoC}, l_{EoC}, def$*, the* integration of parameter choice *yields a program*

$$P_c = (\mathbf{X} \uplus \mathbf{X}_{par}, \mathbf{X}_{in}, (L \uplus L_c, E \uplus E_c), l_{EoC}, l_{x_1}, def)$$

where the parameters are denoted by $x_1, \ldots, x_n = \mathbf{X}_{par}$*, and*

- $L_c := \{l_x \mid x \in \mathbf{X}_{par}\}$ *are new program locations in between which the choice of retain variables occurs – one for each parameter,*
- $E_c := \{(l_{x_1}, x_1 :=?, l_{x_2}), \ldots, (l_{x_n}, x_n :=?, l_{EoC})\}$ *are the actual nondeterministic choices for every parameter.*

Figure 6 illustrates the result of integrating parameter choice into our running example, assuming immediate write semantics. Note that, as in Fig. 5, we use sequences of assignments instead of putting them on separate edges to avoid clutter. This time, similar to the reinitialising assignments, we prepend a sequence of havoc-instructions that realise the nondeterministic choice of a retain configuration, before the actual program semantics are considered. To check whether a retain configuration exists s.t. the program is restart-robust w.r.t. the invariant $\varphi(\mathbf{X})$, it suffices to formalise the program in the usual way (cf. Sect. 3.2) and check whether the CTL specification $\mathsf{EX}\ \mathsf{EX}(pc = l_{\text{EoC}} \rightarrow \varphi(\mathbf{X}))$ holds.

With the illustrated approach, the CTL formula will always need as many EX as parameters are present, to quantify over the prepended path up to l_{EoC}. Note that in practice, this sequence of choices will usually be characterised as a

single composite choice, and a single EX will suffice. Unfortunately, as with the CHC-based modelling, switching to CTL will result in more general procedures being used by a verifier.

5 Counterexample-Guided Synthesis of Safe Retain Configurations

Due to the need for existential quantification in parameter synthesis, a reduction to the previous formalisms will result in significantly more complex decision procedures being used. However, our use of existential quantification is very specific in that we only quantify over Boolean variables and their values also stay constant throughout the possible executions. Therefore it seems natural to manage the choice of parameters oneself, and reuse the efficient procedures for reasoning about restart-robustness for fixed parameters.

Counterexample-guided abstraction refinement (CEGAR) [15] is a general framework for computing an over-approximation, by finding counterexamples that reveal issues with the current approximation and improving it w.r.t. them iteratively. Similar to [13], we use this scheme to over-approximate the supposedly "safe" choices for parameters and refine them iteratively, until all that remains is a characterisation of choices that are guaranteed to exhibit only restart-robust behaviour.

In Sect. 4.1 we have seen that the universally quantified CHCs of our parametrised program check whether all parameter choices lead to restart-robust behaviour. If we had a guess at a characterisation $\mathit{safe}(\mathbf{X}_{\mathrm{par}})$ of safe choices, the same machinery could be used to prove that all these choices indeed result in restart-robustness, by checking whether our CHCs are satisfiable in the context of the following query

$$\varphi(\mathbf{X}) \leftarrow p_{\mathrm{EoC}}(\mathbf{X} \uplus \mathbf{X}_{\mathrm{par}}), \mathit{safe}(\mathbf{X}_{\mathrm{par}}). \tag{7}$$

If no satisfying interpretation of predicates exists, *safe* is a wrong guess and the CHC solver will provide a counterexample that describes a run through the CFA to an end-of-cycle location where φ is violated, and in particular yield the concrete Boolean parameters that led to this. For the next iteration, one would improve *safe* by excluding the apparently bad choice from it.

Algorithm 1 shows the pseudocode of our procedure that follows this intuition. To begin with, the already instrumented and parametrised program P is characterised in terms of both universally quantified CHCs and a symbolic transition system, as presented in Sects. 3.3 and 3.2 respectively. Remember that in the symbolic transition system the program counter is part of the variables, and its $\mathbf{V} = \{pc\} \cup \mathbf{X} \uplus \mathbf{X}_{\mathrm{par}}$ should not to be mistaken for the $\mathcal{V} = \mathbf{X} \uplus \mathbf{X}_{\mathrm{par}}$ used in the CHCs.

The main loop, starting in line 4, implements the refinement procedure described above. If no counterexample is found, *safe* already characterises the retain configurations that lead to restart-robust behaviour. Note that the

Algorithm 1. SynthRetainConf(P, φ)

Input : Program $P = (\mathbf{X} \uplus \mathbf{X}_{\text{par}}, \mathbf{X}_{\text{in}}, \mathcal{A}, l_{\text{EoC}}, l_{\text{EoC}}, \mathit{def})$ with prametrised retains
Predicate $\varphi(\mathbf{X})$ characterising safe states
Variables: Predicate $\mathit{safe}(\mathbf{X}_{\text{par}})$ charactering parameters that do not lead to violations
Universally quantified Horn clauses $\mathcal{H}$

```
1  H ← toHorn(P)                                      // Represent program as ∀CHCs
2  (V, I, T) ← toSymTS(P)                             // and as symbolic transition system
3  safe(X_par) ← true                                 // All parameters are assumed to be safe
4  while ¬sat (H ∪ {φ(X) ← p_EoC(X ⊎ X_par), safe(X_par)}) do      // ∃ violating run?
5      k ← length of violating run
6      c_par ← cube of chosen (Boolean) parameter values in violating run
7      foreach lit in c_par do
8          c̄_par ← c_par with negated lit                           // Flip literal
9          if sat (I(V) ∧ ⋀_{0≤i<k} T(V_i, V_{i+1}) ∧ c̄_par ∧ ¬φ(X_k)) then   // Still violating?
10             c_par ← c_par \ lit                                  // Drop literal
11     safe(X_par) ← safe(X_par) ∧ ¬c_par                           // Block unsafe parameters
12 return safe(X_par)                 // (Potentially empty) region of safe parameters
```

returned predicate will characterise an empty set if no such configuration exists. However if a counterexample exists, we determine its length and the chosen retain configuration, as a conjunction of literals, to prevent the same choice being taken in future iterations (cf. line 11).

To avoid enumerating every single counterexample, CEGAR schemes usually generalise the found counterexample, s.t. a set of counterexamples that violate the specification for the same reason can be excluded at once. Unlike in the general setting of parameter synthesis [13], we do not need special quantifier elimination procedures for the theories that the other variables are represented in, but can adapt generalisation strategies for Boolean cubes.

Similar to the directed but expensive approach of explicitly trying to remove the literals one by one [17], the loop starting in line 7 iterates over every literal once and probes whether it affects the reachability of a violation. To this end, knowing the length of the counterexample, we construct a *bounded model checking* (BMC) [6] instance that characterises the possible executions up to a violation at this length (cf. line 9). To allow for the variables' values to change between different steps, the BMC query uses several instances $\mathbf{V}_i$ of the variables, and $\mathbf{V}_0$ simply denotes $\mathbf{V}$.

Every literal from the cube c_{par}, that characterises the unsafe choice, is then iteratively flipped to determine its impact on the violation. If flipping a literal still leads to a violation, the literal is irrelevant and removed from the cube during iteration (cf. line 10). Note that we use set operations on cubes, like set difference or the subset relation, to denote the operations on the cubes' sets of literals.

Theorem 1 (Generalisation is Sound). *The proposed generalisation procedure always yields a cube $c_g \subseteq c_{par}$ that characterises only unsafe choices.*

Proof. Since the procedure only removes literals from c_{par} to acquire the resulting cube, the relation $c_g \subseteq c_{\text{par}}$ holds by construction. It remains to prove that c_g contains unsafe choices only, i.e. for all choices characterised by c_g, a violating run of length k exists:

$$\underset{\mathbf{X}_{\text{par}}}{\forall} c_g(\mathbf{X}_{\text{par}}) \rightarrow \underset{\mathbf{V} \setminus \mathbf{X}_{\text{par}}, \mathbf{V}_1, \ldots, \mathbf{V}_k}{\exists} I(\mathbf{V}) \wedge \bigwedge_{0 \leq i < k} T(\mathbf{V}_i, \mathbf{V}_{i+1}) \wedge \neg\varphi(\mathbf{X}_k). \quad (8)$$

Base case: The c_{par} that the generalisation is entered with, characterises a single choice that can lead to a violation, so for $c_g = c_{\text{par}}$ formula (8) holds trivially.

Inductive step: Let the formula hold for some c_g. Flipping a literal *lit* in c_g yields $\bar{c}_g$, and two outcomes for the BMC query with $\bar{c}_g$ have to be considered:

- If the query is satisfiable, a violation is reachable even with $\neg lit$ instead of lit. Since both c_g and $\bar{c}_g$ apparently characterise unsafe choices, the formula still holds for their disjunction $c_g \vee \bar{c}_g$, which simplifies to the $c_g \setminus lit$ that we keep.
- If the query is unsatisfiable, $\bar{c}_g$ is a safe configuration and c_g, for which the formula is known to hold, will not be modified.

In fact, this approach is an *anytime algorithm*, since no matter in which order the literals are probed, the formula always stays valid and generalisation can be stopped at any time.

While a single iteration over all literals is not guaranteed to yield the most general form, it already has a significant impact and is cheaper than repeating the procedure until a fixed-point is reached.

6 Experiments

Implementation Details. We implemented Java-prototypes of both the reduction-based and counterexample-guided approach, using the publicly available SMT solver Z3 [29] and the Arcade.PLC platform for analysis of PLC software [5]. Unlike the presented characterisation of single instructions, we implement the common approach of encoding the whole execution cycle as one step [3,8], which is required for efficient reasoning [4].

We use nuXmv [12] and Z3 as off-the-shelf verifiers for the SMV and CHC formalisms that we reduce the verification tasks to. Although the analysed programs do not feature complex operations on bitvectors, and could as well have been modelled with unbounded integers, we characterise the semantics through the theory of fixed-size bitvectors since nuXmv does not support CTL checking over infinite domains, and to the best of our knowledge, no other SMV-based verifier does either.

Due to a bug in the latest version (4.6.0) of Z3 which causes segmentation-faults on retrieval of certain counterexamples, the prototype of our guided approach is linked with an older version (4.5.0) which does not feature CHC-solving with Spacer [23] yet but uses the usually slower *Property Directed Reachability*

(PDR) [21] instead. Since our approach is agnostic about the employed CHC solving procedure, the switch amounted to changing a single parameter.

Furthermore, we do not construct the BMC instance in Algorithm 1 anew in every iteration, but reuse the same one in an incremental fashion and realise the probing for violations by *solving under assumptions* [18].

Benchmarks. The PLCopen is an organisation which drives standardisation and technical specifications in automation. The *PLCopen Safety* library is a collection of such specified modules for domain-specific problems, e.g. how to realise a safe emergency shutdown. We experimented with two groups of PLC programs from this library, whose sizes range from 117 to 1450 program locations per cycle. Programs from [31] are elementary modules, each one implementing a particular safety concept, while [32] features user examples which combine these to form more complex applications.

The considered specifications are invariants that have been used in other case studies [8] and were either formulated by the PLCopen or derived from their technical specifications – the concept is applicable to all specifications that can be reduced to reachability checking though. The benchmark encompasses 56 specifications, 37 of which concern the elementary modules, while the remaining 19 refer to the composite applications.

These programs were not designed with restarts in mind, so we investigate whether they happen to nevertheless be restart-robust w.r.t. the specifications, and whether safe retain configurations exist at all. Since the elementary modules exhibit state-machine semantics, featuring a `DiagCode` variable that tracks the current mode of operation, we declare it to be a retain variable beforehand – similar to `fs` in our running example.

Since the encoding of an execution cycle as a single step is negligibly fast and needs to be performed only once for each program, independent of the checked specification or chosen backend-verifier, we only compare the CPU time spent by the verifiers to allow for a direct comparison of the techniques.

All experiments were performed on a 64 bit Linux machine with 3.5 GHz, 8 GB of RAM and a timeout of 1800 s. They can be reproduced with the artefacts available on our website[1]. Note that for clarity, this package also features analysis results of the running example, and the actual CFAs and their encodings.

Results of Restart-Robustness Checking. In the following, we discuss the measurements for our experiments on verifying restart-robustness w.r.t. a given specification, using the formalisation presented in Sect. 4. To this end, we measure the time it takes to check a specification on the original program, treating retain variables like regular ones, and the time spent on delayed and immediate write instrumented variants of the program. Overall this results in 168 verification tasks.

[1] https://arcade.embedded.rwth-aachen.de/ifm18_restart.tar.gz.

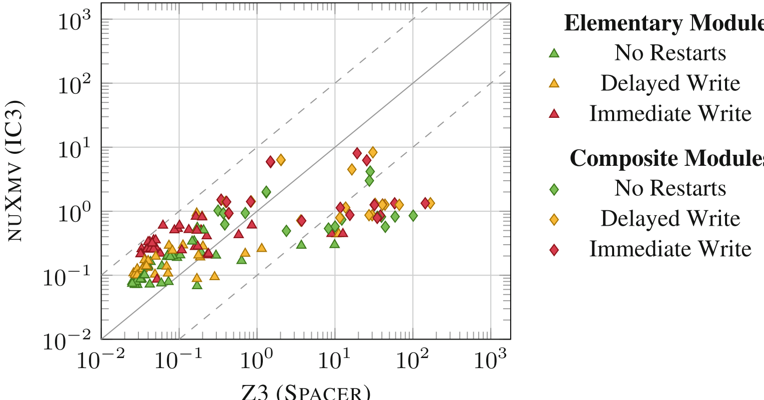


Fig. 7. Time [s] spent on checking restart-robustness w.r.t. each specification and semantics (Color figure online)

Figure 7 compares the runtime of the state-of-the-art tools nuXmv and Z3 on these tasks, with the underlying verification procedures IC3 [11] and Spacer, for the SMV and CHC formalism respectively. The colouring of the marks encodes which restart semantics were considered, while their shape indicates whether the analysed program was elementary or composite.

In our experiments both backends managed to perform all verification tasks in the given time. At a first glance, what strikes the eye, is that nuXmv was about an order of magnitude faster than Z3 on many of the composite examples, while the elementary modules were mostly analysed in less than a second by both tools. However, since this does not apply to all verification tasks for composite modules, compositionality does seem not to be the relevant point here. On closer inspection, we found that in all of the cases where Z3 performed worse, no satisfying interpretation of the CHCs existed. While this is hardly noticeable for the easier tasks, it becomes more apparent in the more complex cases. This might be attributed to nuXmv being more tuned for reasoning over bitvectors, thus quicker to identify unsatisfiable instances, or Spacer not being able to play it's strengths given the non-compositional encoding of program semantics [9,22].

According to the distribution of colours in Fig. 7, the additional consideration of different variants of restart semantics does not seem to have a significant impact on the verification times. The approximate clustering into tasks on elementary and composite programs suggests, that the complexity of the examined program is still the deciding feature. Taken as a whole, the results show that this approach to modelling and verification of restart-robustness w.r.t. some invariant is indeed reasonably fast and feasible.

Results of Synthesis. In this section, we discuss the measurements for our experiments on synthesis of retain configurations that realise restart-robust behaviour w.r.t. a given specification. Since synthesis without retain variables

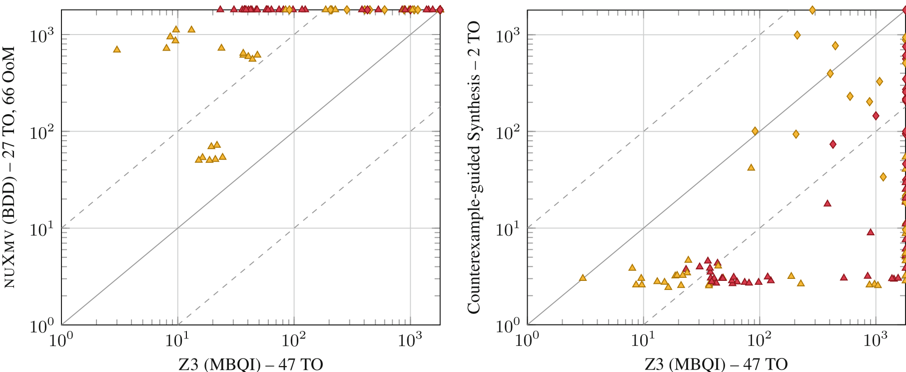


Fig. 8. Time [s] spent on synthesis of restart-robust configuration for each spec and semantics (Color figure online)

does not make sense, we consider the specifications only in the context of delayed or immediate write instrumented and parametrised programs, as seen in Sect. 4.1.

The plots in Fig. 8 illustrate our measurements of the time the verifiers spent on each of the 112 verification tasks, reusing the notation from Fig. 7. To begin with, we focus on the left one, that again compares the runtime of nuXmv and Z3, which now resort to more general decision procedures, i.e. BDD-based CTL checking [27] and a variation of *model-based quantifier instantiation* (MBQI) [19,33] respectively.

Unfortunately, purely BDD-based verification does not scale well for these programs [8], causing nuXmv to run out of memory for 66 verification tasks. In the plot these cases are visualised as timeouts too, i.e. the runtime is set to 1800 s even though the running out of memory occurred earlier. While all synthesis tasks for the composite programs ran out of memory, 27 tasks for the elementary programs caused proper timeouts. Only 19 tasks, all of which targeted elementary modules, could be performed within the resource limits.

Z3 turned out to be significantly more useful for parameter synthesis, timing out only 47 times, and never running out of memory. In contrast to nuXmv, it even manages to determine whether safe retain configurations exist for 14 specifications for the composite programs, and only times out in 11 cases for the elementary ones. We can also observe that, in contrast to plain checking of restart-robustness w.r.t. some property, the type of instrumentation has an impact on the time needed for parameter synthesis. For example, looking at the fastest runs of Z3 we find only yellow triangles, that is tasks on delayed write instrumented elementary modules, ranging from 3 to about 30 s. The corresponding cluster of red triangles, for the immediate write instrumented variants, ranges from about 20 to 150 s.

Nevertheless these results suggest, that the established verification pipelines for checking reachability properties are not practical for parameter synthesis. This observation originally motivated us to devise our own procedure.

Let us now consider the right plot of Fig. 8, which compares the counterexample-guided technique from Sect. 5 with Z3's approach. It is easy to see that our guided approach performs significantly better – often even by more than an order of magnitude. Although it still times out for two of the specifications that refer to the biggest program in our benchmark, the remaining 110 synthesis tasks finished in time. Furthermore, the fact that immediate write instrumented programs yield more complex synthesis problems, does not seem to have a noticeable impact on the runtime. In particular, the clustering of delayed and immediate write instrumented elementary modules that is visible in the x-coordinates, is not apparent in the y-coordinates.

Looking at the plot one might also notice the clustering of data points right above the 2 s mark for our approach. This is due to the fact that we measure the runtime of a verifier from start to finish, i.e. not just the verification procedure, and since our procedure builds upon Arcade.PLC, the first few seconds of every run are spent on the JVM starting, the PLC program being compiled, and the execution cycle being characterised as a single step.

It is interesting to see that although Z3's approach was worse overall, it managed to analyse one of the cases where our technique timed out. In the end, only one verification task remains unsolved by all approaches.

7 Conclusion

While retain variables were introduced with better safety in mind, they allow for subtle corner cases and unexpected behaviour that only occurs after program restart. We are the first to formalise a logic controller's restart behaviour in the context of delayed or immediate write semantics for retain variables, and approach verification of a program's restart-robustness w.r.t. a specification.

To aid in the design of restart-robust software, we illustrated how synthesis of safe retain configurations can be reduced to verification conditions for existing tooling. We also proposed a counterexample-guided procedure which incrementally approximates a region of safe retain configurations, by exploiting the fact that the actual parameters of the formalisation are Boolean, independent of the retain variables' types. Our experimental results show that the verification conditions for restart-robustness can be solved by established tooling in reasonable time. However, synthesis was only feasible when approached with the counterexample-guided technique.

Future Work. There are several ways in which we want to deepen this line of research. On the one hand we want to examine the feasibility of obvious optimisations, like employing an encoding that allows for compositional reasoning [9,22] or looking into more sophisticated generalisation schemes. This alone might suffice to eliminate the last two timeouts in our benchmark.

On the other hand, we plan on investigating whether a definition of restart-robustness as a relational property between the nominal and restart-augmented behaviour is practical, i.e. given no specification but the program's nominal behaviour, to what extent may the restart-augmented behaviour deviate from it to still be considered robust?

References

1. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE 2005, Lisbon, Portugal, 5–6 September 2005, pp. 82–87 (2005)
2. Batt, G., Page, M., Cantone, I., Goessler, G., Monteiro, P.T., de Jong, H.: Efficient parameter search for qualitative models of regulatory networks using symbolic model checking. Bioinformatics **26**(18), i603–i610 (2010)
3. Beckert, B., Ulbrich, M., Vogel-Heuser, B., Weigl, A.: Regression verification for programmable logic controller software. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 234–251. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25423-4_15
4. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, Austin, Texas, USA, 15–18 November 2009, pp. 25–32 (2009)
5. Biallas, S., Brauer, J., Kowalewski, S.: Arcade.PLC: a verification platform for programmable logic controllers. In: IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, Essen, Germany, 3–7 September 2012, pp. 338–341 (2012)
6. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
7. Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II. LNCS, vol. 9300, pp. 24–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2
8. Bohlender, D., Hamm, D., Kowalewski, S.: Cycle-bounded model checking of PLC software via dynamic large-block encoding. In: SAC 2018: Symposium on Applied Computing, Pau, France, 9–13 April 2018 (2018, to appear)
9. Bohlender, D., Kowalewski, S.: Compositional verification of PLC software using horn clauses and mode abstraction. In: 14th International Workshop on Discrete Event Systems, WODES 2018, Sorrento Coast, Italy, 30 May–June 1 2018 (2018, to appear)
10. Bohlender, D., Simon, H., Kowalewski, S.: Symbolic verification of PLC safety-applications based on PLCopen automata. In: MBMV 2016, pp. 33–45 (2016)
11. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
12. Cavada, R., et al.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22

13. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Parameter synthesis with IC3. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013, pp. 165–168 (2013)
14. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0025774
15. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15
16. Darvas, D., Majzik, I., Blanco Viñuela, E.: Formal verification of safety PLC based control software. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 508–522. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_32
17. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: International Conference on Formal Methods in Computer-Aided Design, FMCAD 2011, Austin, TX, USA, 30 October–02 November 2011, pp. 125–134 (2011)
18. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electron. Notes Theor. Comput. Sci. **89**(4), 543–560 (2003)
19. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_25
20. Hauck-Stattelmann, S., Biallas, S., Schlich, B., Kowalewski, S., Jetley, R.: Analyzing the restart behavior of industrial control applications. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 585–588. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_38
21. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_13
22. Komuravelli, A., Bjørner, N., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using horn clauses over integers and arrays. In: FMCAD 2015, pp. 89–96 (2015)
23. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in SMT-based unbounded software model checking. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 846–862. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_59
24. Koskinen, E., Yang, J.: Reducing crash recoverability to reachability. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 97–108 (2016)
25. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View. Texts in Theoretical Computer Science. An EATCS Series, 2nd edn. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-540-74105-3
26. Manna, Z., Pnueli, A.: Temporal verification of reactive systems - safety. In: Broy, M. (ed.) Program Design Calculi, vol. 118, pp. 287–323. Springer, Heidelberg (1995). https://doi.org/10.1007/978-3-662-02880-3_10
27. McMillan, K.L.: Symbolic Model Checking. Kluwer, Dordrecht (1993)

28. Moon, I.: Modeling programmable logic controllers for logic verification. IEEE Control Syst. **14**(2), 53–59 (1994)
29. de Moura, L.M., 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). https://doi.org/10.1007/978-3-540-78800-3_24
30. Ovatman, T., Aral, A., Polat, D., Ünver, A.O.: An overview of model checking practices on verification of PLC software. Softw. Syst. Model. **15**(4), 937–960 (2016)
31. PLCopen TC5: Safety Software Technical Specification, Version 1.0, Part 1: Concepts and Function Blocks. PLCopen, Germany (2006)
32. PLCopen TC5: Safety Software Technical Specification, Version 1.01, Part 2: User Examples. PLCopen, Germany (2008)
33. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: Efficiently solving quantified bit-vector formulas. Formal Methods Syst. Des. **42**(1), 3–23 (2013)

Efficiently Characterizing the Undefined Requests of a Rule-Based System

Zheng Cheng[1(✉)], Jean-Claude Royer[2], and Massimo Tisi[2]

[1] Inria Rennes - Bretagne Atlantique, LS2N (UMR CNRS 6004), Rennes, France
zheng.cheng@inria.fr

[2] IMT Atlantique, LS2N (UMR CNRS 6004), Nantes, France
{jean-claude.royer,massimo.tisi}@imt-atlantique.fr

Abstract. Rule-based systems are used to define complex policies in several contexts, because of the flexibility and modularity they provide. This is especially critical for security systems, which may require to compose evolving policies for privacy, accountability, access control, etc. The inclusion of conflicting rules in complex policies, results in the inability of the system to unambiguously answer to certain requests, with possibly unpredictable effects. The static identification of these undefined requests is particularly challenging for unconstrained rule-based systems, including quantifiers, computations and chaining of rules. In this paper we introduce a static method to precisely characterize the set of all undefined requests for a given unconstrained rule-based system, providing the user with a global view of the rule conflicts. We propose an enumerative approach, made usable in practice by two key performance optimizations: a finer classification of the rules and the resort of the topological sorting. We demonstrate its application on a well-known policy with more than fifty rules.

1 Introduction

Rule-based systems are widely used in very different contexts, ranging from knowledge representation and reasoning to system configuration, from logic programming to databases. Among these contexts, security systems are especially witnessing a significant growth in production of critical, complex and rapidly evolving rule-based policies aiming to offer strong guarantees (of privacy, accountability, etc.) in modern networking environments (including Internet of Things, Software-Defined Networks, etc.). A rule expresses in a concise and natural manner the link between some conditions and a conclusion. This `if then else` semantics is familiar to many software stakeholders, and allows for the definition of modular systems and their flexible evolution.

In this paper we consider a strict logical context, where a rule system is a finite set of logical implications and in conjunction with a request it ensures a reply. Efficient methods for verifying the correctness of such systems in practice is an important research subject [1]. We are particularly interested in one type of error, namely rule conflicts, that cause some requests to be *undefined*, i.e. to have

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 69–88, 2018.
https://doi.org/10.1007/978-3-319-98938-9_5

several incompatible replies. The risk of rule conflicts is very relevant in modern security systems, that often compose independent and evolving policies. We are here interested in a general notion of conflicts entailing the system execution and leading to a runtime bug. We are not focusing on redundancies, misconfigurations or other similar problems which can be considered either as simplifications or holes in the rule system.

There are already several verification methods and algorithms for rule-based policies, in expert systems and databases [2], for Web policies and contracts [3], and in the security domain [4–6]. In this paper we will focus on formal methods and assume a formal policy written in a decidable logic. Moreover we will consider unconstrained systems with complex conclusions and *chaining* of rules (*i.e.* the conclusion of one rule can be used to match other rules and produce new derivations). For these systems we want to provide a *precise characterization* of the set of all undefined requests the user can present, that is in general an infinite set. This characterization constitutes a global view of the conflicts in the system, and a valuable aid in debugging extensive rule sets.

Techniques that have the potential to check for conflicts in unconstrained systems with chaining, divide in two categories. The *testing method* (e.g., [7,8]) computes the set of undefined requests by generating large sets of ground sentences as test requests, and checking the unsatisfiability of each of them in conjunction with the rule system. The testing method suffers from two main drawbacks: *(i)* the global cost of the request generation and evaluation is very high and *(ii)* the test set has finite coverage over the infinite set of undefined requests. The *verification method* (e.g., [9–12]) considers consistency properties (e.g., it is not possible to deny and permit an access at the same time) and tries to prove these properties. It has similar drawbacks since it is generally costly, and does not aim at an exhaustive view of all possible conflicts.

Our contribution is to provide an enumerative method based on symbolic manipulation of the rules and a satisfiability procedure to exhaustively find the precise set of undefined requests in a rule system. The computation complexity is exponential, but we provide two optimization steps to enhance its practical applicability: an iterative method with rule classification and a sorting algorithm. Finally, we evaluate our approach on a well-known case study in XACML, translated into FOL. The experimentation shows that our method is suitable for the verification of rule systems of this size (*i.e.*, 47 rules), where in less than one hundred seconds it produces and analyzes less than one thousand new rules (summarizing the analysis of 2^{47} rule combinations).

The content of this paper is structured as follows. Section 2 describes related work in the area of rule-based systems and checking for conflicting rules. Section 3 presents a motivating example. Section 4 provides the necessary background and definitions to understand our approach. Section 5 illustrates our enumerative method and optimizations. In Sect. 6 we evaluate our method, based on a well-known case study. Lastly, In Sect. 7 we conclude and sketch future work.

2 Related Work

An extensive literature studies the management of rule-based systems. The survey [1] shows that verification and validation of rule-based systems in practice is mainly based on testing or code review which are of course not sufficient to prove that a system is free from bugs. Validation and verification techniques for various kinds of rule-based systems have also been discussed in [2] for expert systems and database management or [3] for Web policies and contracts. In the domain of security policies, the problem of conflicts has been intensively studied. In surveys on security [4–6], conflict detection is a central problem but it is typically treated together with other tasks like finding bugs, redundancies, misconfigurations, etc. We propose a specific solution for static conflict detection, that we believe to be one of most critical issues in modern rule-based systems.

There are already some efficient algorithms to statically detect conflicts in access control policies [13–18]. The method is generally to look for conflict in all combinations of a number of rules (often only two rules). However, most of these methods support only policies with discrete conclusions (like permit and deny) and even only handle discrete attributes as conditions (ABAC policies). Our approach is more general because, except decidabiblity of satisfiability, we do not assume constraints on the rule system, as we could have conditions, functions, any kind of conclusions, possibly with quantifiers and free variables. Even with unbounded rule systems we claim to generate all the undefined requests of the system.

In case of complex systems with predicates in conclusions and allowing the chaining of rules, the existing solutions are not numerous. There are approaches based on testing, used in different contexts, e.g. [7,8,16,19]. A few studies rely on checking satisfiability as in [20], but it is too weak since satisfiability is always a system requirement. Other formal methods and verification, for instance [9–12], use manual proof or derivation tools and are able to prove expected properties of the system (e.g., an access is never both permitted and denied). In practice these properties concern a finite set of authorizations, but in case of unrestricted obligations and chaining of rules the enumeration of the property of interest can be an issue. We do not expect to compete with these approaches on the side of efficiency in detecting a single conflict. Differently from these works, our method automatically computes all the conflicting requests of the system, even if it is not a finite set.

In the domain of constraint solving, we think that techniques for extracting *minimal unsat core* and *maximal sat core* are closely complementary to our work (e.g. [21,22]). We concretely show in this work how to encode the exhaustive conflict detection problem of security policies as a satisfiability problem. To make sure that the problem can be solved within reasonable time, we introduce performance optimizations as symbolic manipulations, before providing the problems to the constraint solver. We do not directly use *minimal unsat core* nor *maximal sat core* techniques to solve our problem. However, our manipulations can be also adapted for enhancing alternative solutions for conflict detection that rely on sat/unsat core (e.g. [23]).

3 Motivating Example

To exemplify our approach, we refer to a very simple example of rule-based policy for a medical center, inspired from [24]. The example is illustrated in Listing 1.1 using the syntax of the Z3 solver, where we represent implication as an infix operator. This example is a first-order example with predicates and free variables but our approach, detailed in the next section, is applicable to any decidable logic extending propositional logic. Here `h`, `p` are universally quantified variables (respectively for hospital personnel and patients) and other terms are predicates or boolean operators. The system decides about read and write access rights to the information of a patient (`pread`, `pwrite`). Hospital personnel comprises the `doctor`, `nurse` and `chief` roles. Personnel can be assigned to the ward of a patient (`sameward`). Despite representing a simple policy, user-defined rules in Listing 1.1 contain conflicts, missing information and redundancies. Briefly, hospital employees with multiple roles can not be assigned to the ward of any patient (rule 1), doctors have read and write access to any patient data (rule 2), nurses can not get information on patients from other wards (rule 3), a doctor can access data for the patients of his ward (rule 4), a chief has read access to any patient data (rule 5).

Listing 1.1. Input for the hospital example

```
1  And(doctor(h), nurse(h)) => Not(sameward(h, p))
2  doctor(h) => And(pread(h, p), pwrite(h, p))
3  And(nurse(h), Not(sameward(h, p))) => Not(pread(h, p))
4  And(doctor(h), sameward(h, p)) => pread(h, p)
5  chief(h) => pread(h, p)
```

We provide an automatic prototype that, given a system like the one in Listing 1.1, identifies all *undefined requests*, e.g. requests having multiple ambiguous answers because of rule conflicts. When provided with Listing 1.1, our prototype produces the output in Listing 1.2.

Listing 1.2. Prototype output for the example

```
[0, 1, 0,-1,-1]
    And(doctor(h), Not(nurse(h))) => And(pread(h, p), pwrite(h, p))
[0, 0, 1, 0, 0]
    And(Not(doctor(h)), nurse(h), Not(sameward(h, p)), Not(chief(h)))
        => Not(pread(h, p))
[0, 0, 0, 0, 1]
    And(Not(doctor(h)), Or(Not(nurse(h)), sameward(h, p)), chief(h))
        => pread(h, p)

#  ----------- Unsafe --------------
[1, 1, 1]
    And(doctor(h), nurse(h), Not(sameward(h, p))) => False
[1, 1, 0]
    And(doctor(h), nurse(h), sameward(h, p)) => False
[0, 0, 1, 0, 1]
    And(Not(doctor(h)), nurse(h), Not(sameward(h, p)), chief(h)) => False
```

The rule system in Listing 1.2 is logically equivalent to the original rule system and composed of pair-wise exclusive rules[1]. Furthermore, each rule is prefixed by a sequence of digits indicating the combination of the original rules which produces this rule. For instance, the first rule in Listing 1.2 comes from the combination of the original rule 2 and negation of rule 1 and 3; -1 is a don't care digit for rule 4 and 5. The rules tagged as unsafe denote indeed set of undefined requests, for instance `And(doctor(h), nurse(h), Not(sameward(h, p)))` is undefined which results from the combination (`[1, 1, 1]`) of the three first rules. That means that its conjunction with the rule system is unsatisfiable. The other rules denote defined and undefined requests. On one side, the defined requests should intersect the conjunction of the rule condition and the rule conclusion. On the other side, undefined requests are included, by implication, in the conjunction of the rule condition and the negation of the rule conclusion. Related to the first not unsafe rule, `And(doctor(h), Not(nurse(h)))` is a defined request and `And(doctor(h), Not(nurse(h)), Not(pread(h, p)))` is an undefined one.

With such information the user can exactly know what are all the conflicting problems, and then what are the requests that this rule system can handle. Furthermore, he can localize the problems in the rule system, by knowing what is the rule combination leading to undefined requests. For instance, the first unsafe rule comes from the combinations of rules 1, 2 and 3. It states that any request about a personnel member that is at the same time a doctor and a nurse will lead to a rule conflict, even if the person is not assigned to the ward of the patient. The conflict involves the first three rules, since chaining rule 1 with respectively rule 2 and 3 leads to contradictory answers, where the data from the patient can and can't be read at the same time (`pread(h, p)` and `Not(pread(h, p))`).

The next section will introduce the required concepts and Sect. 5 will present the principles of our algorithm.

4 Background and Definitions

In this section, we define a terminology that will be consistently used in the rest of the paper. Our focus are systems that process requests and produce replies. One important problem is due to requests leading to evaluation failures and often called conflict in the literature. Here we formalize the notion by defining the term *undefined request*. We focus on *rule systems*: a conjunction of rules, a rule is $A => B$ with A and B in some logical language. Considering that satisfiability decision has made important progress, we expect to build a new management method of rule-based systems by reusing efficient tool support (e.g., Z3, SMT, SPASS, TSPASS).

Let R be a satisfiable policy system with its *set of expected requests* REQ. REQ is a finite set of satisfiable logical formulae that we are interested in answering. It represents the set of expected inputs to the policy system. Requests and

[1] To improve readability, we simplify the actual output from our prototype in Listing 1.2. The complete output and our prototype can be found in https://github.com/atlanmod/ACP.

replies are satisfiable logical expressions, they could be ground (without variables) or containing free variables (implicitly universally quantified) or quantified variables. Indeed all the expressions (requests, replies, conditions, conclusions, ...) are assumed to be written in a given logical language which may allow variables, quantifiers, modal operators, and so on, providing it has a satisfiability procedure.

Definition 1 (Rule-based system). *A rule-based system (*R*) consists of a finite conjunction of rules noted $R = r_{1 \leq i \leq n}$, where each rule r_i takes the format of D_i => C_i, where D_i stands for the premise/condition of the rule, and C_i stands for conclusion of the rule. In addition we assume that R is satisfiable and does not contain tautological rules.*

We should note that a rule-based system is interesting if it is satisfiable and not valid. This means that it can derive non trivial facts from the request. Thus we assume that R is satisfiable and it does not contain tautological rules, that is $D_i \wedge \neg C_i$ is satisfiable for all i. A request will be a satisfiable logical expression submitted to a rule-based system and leading to a reply, another logical expression. Note that a request should trigger at least one rule, otherwise its conjunction with the system is satisfiable but does not infer a reply. Thus we make the natural hypothesis that our requests are satisfiable and imply $\bigvee_{1 \leq i \leq n} D_i$. We will say that a set of rules is *exclusive* if their conditions are pair-wise disjoint.

Definition 2 (Request evaluation in rule-based system). *Let req be a logical expression, evaluating it against a given rep, called the reply, means that $req \wedge R$ => rep is valid or equivalently $req \wedge R \wedge \neg rep$ is unsatisfiable.*

We are interested in the evaluation problem raised due to undefined requests.

Definition 3 (Undefined requests). *A req request is said undefined if and only if req and R are both separately satisfiable but $req \wedge R$ is unsatisfiable.*

An undefined request in fact leads to a problem since its evaluation leads to multiple incompatible replies. This definition is stricter than satisfiability and covers the usual notion of conflicts we found for instance in security policies [6,25]. This has two simple consequences: *(i)* any useful system has undefined requests and *(ii)* these undefined requests are included in $\bigvee_{1 \leq i \leq n} D_i$. Furthermore any rule ($D_i$ => C_i) which is satisfiable and not valid has *1-undefined* requests, that is requests invalidating this rule alone (in other words requests which imply $D_i \wedge \neg C_i$). In this work, we aim to study a complete and efficient algorithm to ensure the safety of a given rule-based system (Definition 4).

Definition 4 (Safety of rule-based system). *A rule-based system with its set of expected requests is safe if and only if it does not contain any undefined requests, that is for all satisfiable request $req \in REQ$ implies $req \wedge R$ is satisfiable.*

As we can see this property assumes that the input system is satisfiable, and it is different from many approaches looking for logical inconsistencies in a system, for instance [9,10,20].

5 Characterizing Undefined Requests

The management of policies and requests requires to consider several activities: looking for the existence of one undefined request, checking a request, finding all the undefined requests, localizing the conflicting rules, resolving the undefined requests and evaluating a request. We here focus on finding all the undefined requests since this global knowledge is necessary to understand the failures in the system and to globally fix them. We do not study the fixing process but we will give a few related hints later. As we will see, one additional benefit of our method is to get localization of conflicting rules for free.

This section shows a decision procedure for the safety property (if satisfiability is decidable), its theoretical complexity is EXPSPACE. Our approach is based on transformations of the original rule-based system. It is important to preserve the conditions and conclusions of rules as they represent the expected requests and replies. Hence we will build new rules by mixing the conditions on the left hand side and conclusions on the right hand side.

5.1 The Enumerative Method with Exclusive Rules

Our enumerative method to compute undefined requests is based on rewriting the original rule system into its equivalent form in terms of exclusive rules.

A minor point of the method is to allow requests with variables. By considering requests with variables rather than ground requests, we aim to cover the whole (sometimes infinite) set of requests. For example, in Listing 1.1, a ground request like `And(doctor(Jack), nurse(Jack), Not(sameward(Jack, May)))` with constants `Jack`, and `May` is undefined. However, there are many other different ground requests that expose this problem. To capture the essence of these problems, we use the requests `And(doctor(h), nurse(h), Not(sameward(h, p)))`, which is a short hand for $(\forall h, p \cdot (doctor(h) \wedge nurse(h) \wedge \neg\ sameward(h, p)))$.

Furthermore, an undefined request is *unsafe* in the sense that it contains (*i.e.*, it is deducible by logical implication from) only undefined requests. However, defined requests are not *safe* in general, in the sense they are not maximally defined.

Definition 5 (Safe request). *Let R be a rule system, a satisfiable request req is safe if and only if any satisfiable request which implies req is defined.*

Second, Lemma 1 establishes that we can rewrite the original rule system into an equivalent form, namely exclusive rules. The lemma can be proved by recurrence on the size of rule system n.

Lemma 1 (Exclusive rules). *Let R be a rule system, we have the equivalence*

$$\bigwedge_{1 \leq i \leq n} (D_i => C_i) <=> \bigwedge_{I} ((\wedge_{i \in I} D_i \wedge_{j \notin I} \neg D_j) => \wedge_{i \in I} C_i)$$

where I is any non-empty subset of $\{1, \ldots, n\}$.

The rewriting of a rule system with n rules generates $2^n - 1$ exclusive rules. The rule system is composed only of pair-wise disjoint rules, and preserves the original set of defined and undefined requests.

For example, the rewriting applied to the rule system shown in Listing 1.1, results in a total of 31 exclusive rules (partially shown in Listing 1.3). As shown in Listing 1.3, first, we abbreviate each rule by its appearance order in Listing 1.1, e.g. *R1* is `And(doctor(h), nurse(h)) => Not(sameward(h, p))`. Second, we use a D function to get the condition part of a rule, and a C function to get the conclusion part of a rule, e.g. *D(R1)* is `And(doctor(h), nurse(h))` and *C(R1)* is `Not(sameward(h, p))`. For instance, in Listing 1.4 we show the non-abbreviated form of the first exclusive rule, corresponding to the first line of Listing 1.3. The condition is obtained by conjunction of the first original rule *R1*'s condition, with the negation of the conditions of the other rules (*R2*–*R5*). The conclusion is simply the conclusion of *R1*.

Listing 1.3. Part of abbreviated exclusive rules generated by our enumerative method for the example shown in Listing 1.1

```
1  And( D(R1), Not(D(R2)), Not(D(R3)), Not(D(R4)), Not(D(R5)) ) => C(R1)
2  And( D(R1), D(R2), Not(D(R3)), Not(D(R4)), Not(D(R5)) ) => And( C(R1), C(R2) )
3  And( D(R1), D(R2), D(R3), Not(D(R4)), Not(D(R5)) ) => And(C(R1), C(R2), C(R3))
4  ... Another 28 rules
```

Listing 1.4. The first exclusive rule of Listing 1.3 in non-abbreviated form

```
1  [1, 0, 0, 0, 0]
2      And(doctor(h), nurse(h)), Not(doctor(h)),
3          Not(And(nurse(h), Not(sameward(h, p)))),
4          Not(And(doctor(h), sameward(h, p))), Not(chief(h))
5          => Not(sameward(h, p))
```

As we previously saw, a rule (if not valid) has always 1-undefined requests and the transformation above builds a system whose undefined requests are disjoint unions of 1-undefined requests.

Lemma 2 (Undefined requests of exclusive rules). *req is an undefined request of an exclusive rule system R if and only it is a disjoint sum,* $req = \bigoplus_{1 \leq j \leq m} req_j$*, where* $\bigoplus$ *is the accumulative exclusive-or operator, and each* req_j *is satisfiable and 1-undefined for a given rule* j*.*

Lemma 2 simply results from the exclusive rules and the partition of *req* into disjoint parts related to the conditions D_i. Thus the set of safe requests is defined as the disjoint union of all the safe requests associated to each rule. For each rule the conjunction of the condition and the conclusion defines a safe request. The maximal safe request is $safe(R) = \bigvee_I \wedge_{i \in I} D_i \wedge_{j \notin I} \neg D_j \wedge_{i \in I} C_i$. A request is safe if included in $safe(R)$, it is defined if it intersects $safe(R)$, and else it is an undefined request. From that the safety property can be checked by the satisfiability of elements in REQ against $safe(R)$.

Discussion. While existing testing methods for rule systems are generally able to show the existence of ground undefined requests, they can't prove that a given (unbounded) rule system is safe. The enumerative way has three main benefits:

- it works with the same complexity in case of finite or infinite set of ground requests while the testing approach is not suitable with infinite sets,
- it does not depend on the set of requests thus it outperforms the testing approach in many non trivial examples,
- it may produce undefined requests with variables, that represent an abstraction of the system problems, with localization for free.

Compared to the verification approaches, our enumeration does not require a large set of interesting properties to prove. Moreover, if the property is not satisfied, in the best cases previous work on verification generates a counter-example. In general, it is not possible to produce a characterization of all the counter-examples as we did here.

On the performance side, the enumerative method implies the computation of all the rule combinations, which has an expensive cost. Furthermore, we check each rule for validity and valid rules are discarded as they do not contribute to the set of undefined requests. The enumerative method can have worse performance than the testing methods in case of propositional rule systems. However, we expect our method to perform better than exhaustive testing of rule systems with variables. For instance, with systems expressed in FOL, the exhaustive testing requires a maximal number of 2^{P*D^V} test cases, where P is the number of predicate occurrences in R, D the size of the domain of arguments and V the number of arguments. The set of test cases is growing quickly, while the number of exclusive rules is only dependent on the number of rules in the input system.

Work on *minimal unsat core* and *maximal sat core* are closely related to our enumerative method [21,22]. For example, after our partition in order to identify all the undefined requests of a rule-based system, we could send each exclusive rule to a solver to extract all the minimal unsat cores. However, for the sake of performance, we only check exclusive rules for satisfiability, since theoretically, the problem of extracting minimal unsat core is harder than checking satisfiability.

5.2 The Iterative Method

We present a first improvement of the enumerative method, called the *iterative method*. It is based on adding iteratively a new rule at each step and eliminating some rules as soon as possible. The principle is based on the following property. Assuming that R' is a list of exclusive rules computed with the enumerative method and r_j a new rule. Note that each rule in R' can be uniquely described by its *binary characteristic* which is a binary integer of length n (where n is the rule size of the initial rule system). Let b a binary integer we can construct a rule of R' in the following manner: the condition is the conjunction of the conditions D_i (respectively $\neg D_i$) if the ith digit of b is 1 (respectively 0). The conclusion is the conjunction of the conclusions of R for which digit is 1 in b. Thus speaking of an exclusive rule or its binary characteristic is equivalent. We will say that a rule i of the input system is active in an exclusive rule if the binary digit at position i is 1.

Proposition 1 (Iterative principle). *Let R' built from the enumerative method, then the enumerative result for $R' \wedge r_j$ is obtained from the binary characteristics of rules in R' and for each b we will get two new binary characteristics $2 * b$ and $2 * b + 1$ plus one single last characteristic equal to 1.*

This results from a simple recursive analysis of the binary characteristics of R' compared to that of $R' \wedge r_j$. That means that if R' has m rules we will get $2 * m + 1$ rule in $R' \wedge r_j$.

Furthermore, the main loop of the iterative method is based on the above principle and two optimizations.

Definition 6 (Obvious rule). *A rule $D => C$ is an obvious rule iff D is unsatisfiable.*

The first optimization is to discard obvious rules during the iterative method (Definition 6). Obvious rules are one specific kind of tautologies. Adding them during the iterative method offers no value, since they will just generate two new obvious exclusive rules and make no difference to the last generated exclusive rule that in negation form. Notice that we keep exclusive rules that are tautologies but not "obvious" during the iterative method. The reason is that they have an impact on the iteration, e.g. affect the last generated rule. An additional optimization is possible here but it is postponed to future work.

Definition 7 (Unsafe rule). *A satisfiable rule $D => C$ is an unsafe rule iff it is equivalent to $D => false$.*

The second improvement of the iterative method is to separate unsafe rules. An unsafe rule implies no defined request matching its condition. Checking it efficiently depends from the logical language used and should take care of quantifiers in case of free variables. If a rule is not unsafe then $(D \wedge C)$ is a safe request and the rule has only 1-undefined requests in $(D \wedge \neg C)$. A consequence of this classification is that now the maximal safe request ($safe(R)$) is computed only from rules that are not unsafe.

The above definition paves the way to a further optimization, which is to check for unsafe rules and to store them separately from the others. Hence we stop processing unsafe rules and present them in the final result (possibly with a shorter binary characteristic w.r.t. not unsafe rules, e.g. see the first two unsafe rules of Listing 1.2). With the iterative method and the classification (obvious, unsafe and not unsafe) we expect to decrease the number of generated rules which is a critical factor of the enumerative approach.

For example, by applying our iterative method on the rule system shown in Listing 1.1, the first rule *R1* is checked for its obviousness and unsafeness, and the checks succeed. Then, *R1* becomes the first exclusive rule and the iteration starts. We show in Listing 1.5 the result of first iteration by adding *R2* to the first exclusive rule. Since *R2* also passes obviousness and unsafeness checks, three exclusive rules are built along with iterative method. Their binary characteristics are [1, 1], [1, 0] and [0, 1] respectively.

Listing 1.5. Exclusive rules generated by our iterative method for the example shown in Listing 1.1 at the 1st iteration

```
1 [1, 1] And( D(R1), D(R2) ) => And( C(R1), C(R2) )
2 [1, 0] And( D(R1), Not(D(R2)) ) => C(R1)
3 [0, 1] And( Not(D(R1)), D(R2) ) => C(R2)
```

At the final iteration, we get a total of 6 exclusive rules (3 unsafe, 3 not unsafe, plus 4 eliminated as obvious), which results in a total of 16 unsat checks. Clearly, the iterative approach performs better than the enumerative approach (that in this example would generate 31 rules, by 31 unsat checks), and we anticipate that the effectiveness of the iterative approach would be more visible on larger examples. Note that the iterative method is a quite general solution which requires a logic extending propositional logic with a decision procedure for satisfiability.

5.3 The Sorting Method

The analysis of relations between rule conclusions can lead to significant performance improvements. We argue that in many practical examples, rule conclusions are built on a set of finite predicates, and several rule systems have some pairs of rules whose conclusions are related by inclusion. If we add in the iterative process a new rule $D_j => C_j$ and if we know that it exists $i < j$ such that $C_i => C_j$ is valid then we can simplify the process using the following principle. Let $cond \wedge D_i => conc \wedge C_i$ be the exclusive rule where rule i is active, in the step before the addition of rule j. By adding rule j, the iterative process would generate two rules $cond \wedge D_i \wedge D_j => conc \wedge C_i \wedge C_j$ and $cond \wedge D_i \wedge \neg D_j => conc \wedge C_i$. Given the inclusion relation between C_i and C_j, instead of these two rules, the two rules simplify in $cond \wedge D_i => conc \wedge C_i$, that was already present before adding rule j. This rule will still be exclusive w.r.t. all the other rules generated during the addition of rule j. In the binary characteristic of this exclusive rule, we insert a -1 at position j, to indicate that the condition of rule j does not matter in this combination. This optimization eliminates two satisfiability checks, but most of all it decreases the number of generated exclusive rules.

With this optimization the size of the result depends on the order of rules. Thus, to take the maximal benefit from these relations, we sort the rules from minimal conclusions to maximal conclusions (w.r.t. the implication relation). We achieve this reordering by a simple adaptation of topological sorting, with a complexity in $O(n^2)$ (where n being number of rules). The result of our topological sorting is that each rule is preceded by all the rules with a smaller conclusion than its proper conclusion. For example, the sorting result for our example shown in Listing 1.1 is `R1, R3, R2, R4, R5`.

5.4 The Algorithm

To sum up, in Algorithm 1 we sketch our process to efficiently compute exclusive rules and classify them as unsafe and not unsafe. Notations used in the algorithm require some explanation.

Algorithm 1. Our algorithm to efficiently compute not unsafe and unsafe requests

```
1: NotUnsafes, Unsafes, Negs, Zeros ← dict(), dict(), True, []
2: procedure Main(R)
3:     R ← sort(R)
4:     BuildExRule(D(hd(R)), C(hd(R)), [1])
5:     Negs ← Negs ∧ ¬D(hd(R))
6:     Zeros ← Zeros ++ [ 0 ]
7:     for each rule r_i ∈ tl(R) do
8:         IterativeTableCtr(r_i)
9:     end for
10: end procedure
11:
12: procedure IterativeTableCtr(r)
13:     prevNotUnsafes ← NotUnsafes
14:     NotUnsafes ← dict()
15:     for each b_i ∈ prevNotUnsafes.keys() do
16:         exrule_i ← prevNotUnsafes[b_i]
17:         if C(exrule_i) implies C(r) then
18:             NotUnsafes ← NotUnsafes.update({ b_i ++ [-1] : exrule_i })
19:         else
20:             BuildExRule(D(exrule_i) ∧ D(r), C(exrule_i) ∧ C(r), b_i ++ [1])
21:             BuildExRule(D(exrule_i) ∧ ¬D(r), C(exrule_i), b_i ++ [0])
22:         end if
23:         BuildExRule(Negs ∧ D(r), C(r), Zeros ++ [1])
24:         Negs ← Negs ∧ ¬D(r)
25:         Zeros ← Zeros ++ [ 0 ]
26:     end for
27: end procedure
28:
29: procedure BuildExRule(d, c, b)
30:     rule ← Rule(d, c)
31:     if not isObvious(d) then
32:         if isUnsafe(d, c) then
33:             Unsafes ← Unsafes.update({ b : rule })
34:         else
35:             NotUnsafes ← NotUnsafes.update({ b : rule })
36:         end if
37:     end if
38: end procedure
```

First, we use D and C to get the condition and conclusion of a rule respectively. Second, we inherit traditional list operators *hd*, *tl* and *++* for head, tail and concatenation of lists. Third, we use *dict* to initialize a dictionary, and use *update* to add a pair to a dictionary, and overload *a[b]* for element access in a dictionary (`i.e.` accessing element whose key is b in a dictionary a). Finally, three predicates specific to our algorithm are *sort*, *isObvious* and *isUnsafe*. They perform a topological sort of a list of rules w.r.t. implication between conclusions (Sect. 5.3), check whether the rule is an obvious tautology (Sect. 5.2), and check whether the rule is unsafe (Sect. 5.2), respectively.

The input of our algorithm is a list of rules, the output are classified exclusive rules. To compute from input to output, we use 4 global variables in our computation (line 1). *NotUnsafes* and *Unsafes* are dictionaries, which are used to keep track of the rules detected so far, that are unsafe and not unsafe. Notice that the key of these two dictionaries is the binary characteristic of each exclusive rule, and the value is the exclusive rule itself. *Negs* is a conjunction of formula that represents negated conditions for rules that previously iterated on. *Zeros* is synchronized with *Negs* to record binary characteristics. These two global variables are used when building exclusive rules in negative form on line 23.

Our iterative construction is performed to categorize rules that are unsafe or not unsafe (lines 2–27). It internally uses our *sorting method* to optimize its efficiency (lines 17–18), and uses *BuildExRule* to interact with the solver to check whether a newly constructed exclusive rule is unsafe (lines 29–38). The whole process offers no surprise w.r.t. what we described in Sects. 5.2 and 5.3.

Once the two lists of rules *NotUnsafes* and *Unsafes* are computed by Algorithm 1, the set of undefined requests is precisely characterized as:

- each unsafe exclusive rule denotes a set of undefined requests, `i.e.` all the requests included by implication in the rule condition,
- each exclusive rule that is not unsafe denotes a set of undefined requests, `i.e.` all the requests included by implication in the conjunction of the rule condition and the negation of the rule conclusion,
- no other undefined requests exist for the original system.

6 Evaluation and Discussion

We implement the three previous methods in Python 3 and we use the z3py interface to interact with the Z3 solver [26]. Our input rules are defined using a class of rules reusing the logical expression defined by Z3. The code is available on our on-line repository[2] with a set of examples. We prove, using the solver, that the original rule system is equivalent to the new generated one.

Our evaluation objective is to experiment with middle size examples to observe if the concrete performances are according to our expectations and suitable for a practical use. In our approach, we rely on the Z3 solver, but the solver

[2] Efficiently characterizing the undefined requests of a rule-based system (on-line). https://github.com/atlanmod/ACP.git.

can be changed as long as it provides a decision procedure for satisfiability suitable for the input rule system. Our method is not limited to pure predicates with free variables, however its success and efficiency depend on the ability of the solver to check such construction.

6.1 The CONTINUE Example

We describe here the case of the CONTINUE A policy already used in [20,27] and dedicated to conference management. This policy[3] is specified in 25 XACML files containing 44 rules. Our objective was not to exactly encode this policy but rather to validate that our optimizations are effective in case of a non trivial example. We deviate from the original CONTINUE example in several ways. First, we consider a pure logical translation and this introduces a difference as already discussed in [10]. We do not take into account the combining algorithms since our objective is to observe the undefined requests occurring in the business rules, not to provide an ad-hoc automatic resolution. We also handle free variables and predicates (unary and binary), while this is not the case in the XACML language.

Our initial rule system has 47 rules, with two types, two free variables, 6 binary predicates and 29 unary ones. We start with this example without additional relations. Figures 1, 2, and 3 depict the results we get with a MacBookPro under El Capitan, 2.5 GHz Intel Core i7 and 16 Go 1600 MHz DDR3 RAM. The resulting times were computed from an average of 10 runs.

In the figures we report the number of rules in the system and a few curves: *correct* is the number of non tautology in the system, *time* the time in second, *exclusive*, *not unsafe*, *unsafe* the number of exclusive, not unsafe and unsafe rules. We process the example by taking the first n rules from an arbitrary ordering. The *correct* curve shows that there is no tautology in this system.

The enumerative experiment (Fig. 1) shows clearly an exponential growth in the number of generated rules and in the processing time. While the performances of the iterative method (Fig. 2) are much higher, we still cannot process the full example in a reasonable time. The sorting method (Fig. 3) provides more interesting results in such a case. CONTINUE has many inclusion relations among conclusions (653 relations), which explains the good performances by topological sorting. While this is not a universal property, in our experience, this is often the case in security systems. This is obvious when conclusions are only permit and deny as in some simple access control policies. More generally, in security we expect to control the possible outcomes of the rules, thus defining a limited set of replies. Each rule can then combined these outcomes and thus revealing relations between conclusions. We easily observe this on several of our examples, but a statistical analysis should be perform to validate this assumption.

We should also note that some rules have a great impact on the behavior of the iterative and sorting methods. This is the case with predicate exclusivity

[3] http://cs.brown.edu/research/plt/software/margrave/versions/01-01/examples/.

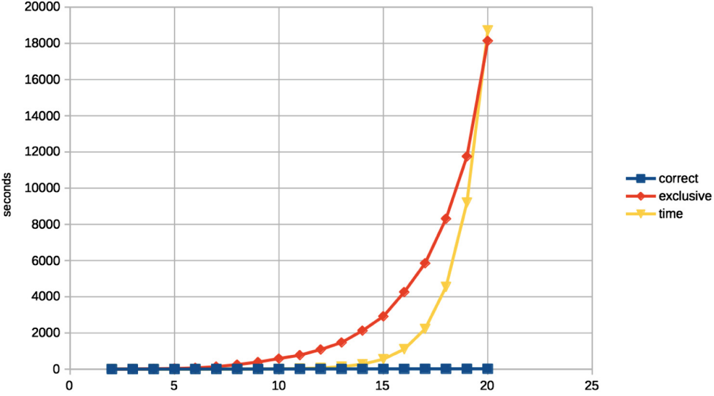


Fig. 1. Enumerative experiments

which is often implicit. Indeed XACML, due to the use of the combining algorithms, does not make hypothesis about the disjunction of roles or permissions of different actions. To observe the stability of our performances we experiment adding a few new rules about roles in the system. For instance, it makes sense to consider that the PC chair is also a PC member and a subreviewer is not a PC member. There are also some relations related to resources, for example there are several different kinds of information about papers. These resources appears only in conditions and alone but never in a conjunction, thus we may consider these resources as disjoint. Adding these rules increases the size of the original system to 57 rules. As shown in the curves in Fig. 4 these disjunctions decrease the number of rules that are not unsafe. With this setting we generate 776 rules (535 unsafe and 241 not unsafe) in nearly 100 s, including the verification of the equivalence with the original system which takes 10 s.

Listing 1.6. An example of conflicting rules

```
// An unsafe case
UNSAFE [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
// The four active  10-13th rules (other negative rules are omitted)
pcchair(X) => pcmember(X)
Or(admin(X), pcchair(X), pcmember(X), subreviewer(X)) => subject(X)
And(PaperAssignments(R), subject(X), isConflicted(X))
       => And(Not(Pread(X, R)), Not(Pwrite(X, R)), Not(Pcreate(X, R)))
And(PaperReviewContent(R), pcmember(X), isEQuserID(X))
       => And(Pcreate(X, R), Pwrite(X, R), Pdelete(X, R))
```

As an example of output in this scenario, Listing 1.6 shows the binary characteristic of one of the first unsafe rules detected (the 10th). As shown by the binary characteristic, this rule denotes a set of undefined requests coming from the composition of the four active rules listed underneath. The four rules are conflicting in a non-trivial way. The next challenge of our work is the automated analysis of each group of undefined requests (like the one in Listing 1.6), for aiding in the resolution of the conflict.

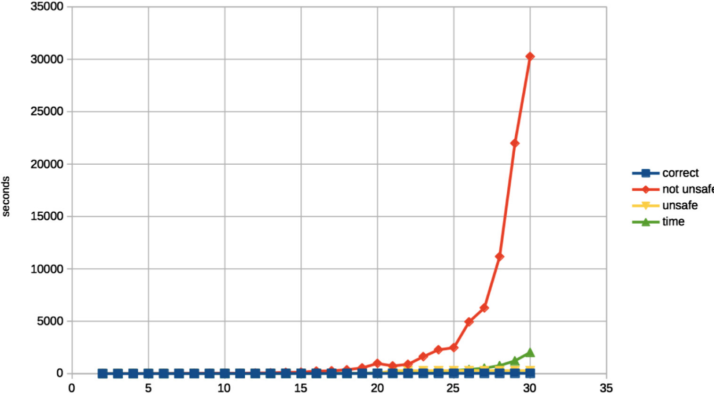


Fig. 2. Iterative experiments

Another usage of this computation is to check if a request is safe, defined or undefined. Starting from the rule classification, our prototype is able to compute the maximal safe request as defined in Sect. 5 then we can check the request against it. Listing 1.7 shows two examples: After the computation of Algorithm 1, both of these examples are processed in less than one second.

Listing 1.7. Two safety tests

```
// Intersecting the maximal safe request it is a defined request
ForAll([X, R], And(Not(PcMember(R)), PaperReviewContent(R), pcmember(X),
                   Not(subreviewer(X)), isEQuserID(X)))
// It  is contained in the negation of the maximal safe request
// thus it is an undefined request
ForAll([X, R], And(PaperAssignments(R), subject(X), isConflicted(X),
                   PaperReviewContent(R), pcmember(X), isEQuserID(X)))
```

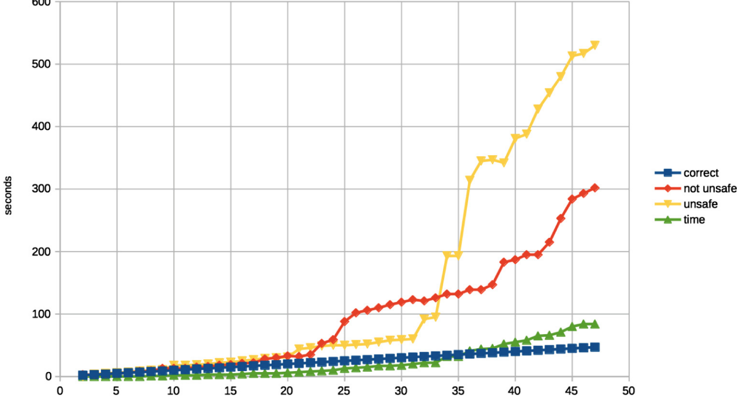


Fig. 3. Sorting implementation

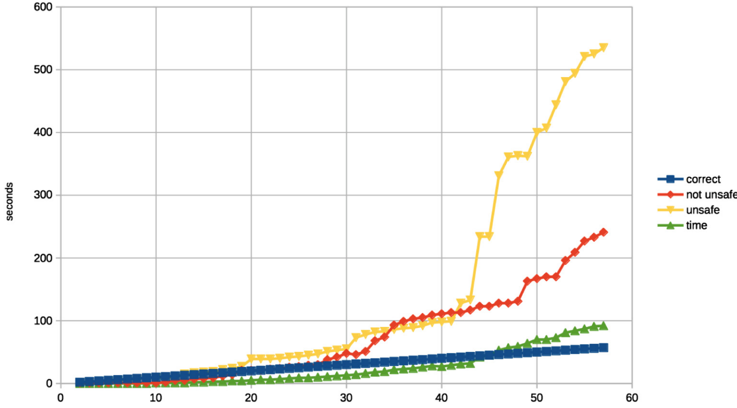


Fig. 4. Sorting test with 57 rules

6.2 Discussion

In summary, through our evaluation, we experiment our approach on a middle-size example, and observe its performances are according to our expectations and suitable for a practical use. There are also some lessons we learned.

Correctness. Decidability of the satisfiability is required for our approach to get an optimal rule classification and best performance. For example, we interact with solver to check conclusion implication, and whether a rule is obvious, unsafe or not unsafe. If the solver cannot give a reply to these questions within a given timeout, an unknown will be returned as result. This would degrade the precision of rule classification result and the performance of our approach. However, when unknown results occur, we always defensively categorize them as not unsafe, and thus will not give incorrect answers to the user.

Usability. It is important to simplify as much as possible the output, to facilitate the inspection of rule conflicts. Some simple cases are already handled in the iterative and sorting steps. For instance, two rules with equivalent conditions are simply merged into one rule with this condition and a conjunction of each conclusion. However, more aggressive simplifications are complex and time consuming. In our current solution, we think that it is better to first produce a result which alerts the user on the presence of undefined requests. In a second step, if the user wants to fix some problems we should provide a simplified version of the rules, and perhaps some hints for resolution of conflicts.

While we do not think that automatic resolution will always match users' expectation, our approach can be extended to suggest some automatic fixes to the user. For instance, the user may want to restrict its set of expected requests to the maximal set of safe requests.

Another idea is to add conditions occurring in the unsafe rules as extra conditions of the not unsafe rules. One approach is to introduce the input rules

one by one and to resolve the generated unsafe cases. If the system has no chaining of rules then there is no further problem and this way will produce a safe system. In the more general chaining case, the user should always cope with 1-undefined requests.

Generalization. While the number of rules is related to the complexity of the rule system, they do not necessarily compromise the generalization of our approach, e.g. the algorithm could take advantage of more obvious rules, or there could have more implications between conclusions.

However, we do agree that more case studies are needed to confirm the generalization for the performance and practicability of our approach. We processed another example[4], consisting of 61 rules for managing resources, hierarchy of roles, permissions and revocation of permissions. The encoding of these rules are more complex than the CONTINUE example, e.g. predicates to represent discrete time. While the relationship between rules are more sparse (259 inclusion relations among conclusions), our sorting method is still much more efficient than the iterative one. For example, during the evaluation process, we observe that iterative method takes about 13000 s to analysis 40 rules in this example. Our sorting method only takes 735 s on the same set of rules. In our preliminary result on this example, we also observe a reasonable growth in its analysis time (5736 s), and find 4 unsafe rules and 17085 not unsafe ones. The last unsafe rule reveals an unexpected conflict due to the hierarchy of resources and not seen in the original description.

Optimization. Currently, our approach produces a logically equivalent system for the input during its analysis. However, we think this restriction can be relaxed, e.g. a new system that is stronger than the original one could still be acceptable for analysis since it guarantees the behavior of the original. Our future work will explore new optimizations based on this kind of relaxation.

Another track of optimization could be decompose input system into subsystems, and pave its way for a map-reduce-flavor algorithm.

7 Conclusion

In this paper we provide a new way to compute all the conflicting problems occurring in a rule based system with chaining of rules. Our methods are rather general since they require a logic extending propositional logic and a decision procedure for its satisfiability. Existing methods rely either on testing or on formal verification but they are not suitable to find the exhaustive set of potential problems. Mixing symbolic manipulations and satisfiability, we provide a decidable enumerative approach to solve this problem but due to its exponential complexity we must provide optimizations. We study two optimizations in order to reduce the number of generated rules: an iterative method with a classification of rules and the use of the topological sorting to take the maximal advantage of

[4] RBAC and ARBAC policies for a small health care facility. http://www3.cs.stonybrook.edu/~stoller/ccs2007/.

relations between rule conclusions. As an evaluation we successfully apply our algorithm to a FOL rule system with more than forty rules. With this instance, rather than computing 2^{47} new rules, we produce less than 1000 rules in less than 100 s. Note that when the rule system is complex, it contains many relations between the predicates, and increases the risk of undefined requests. However, in this case our method, especially the sorting optimization, is particularly efficient. Our automatic method is able to handle middle size examples and more improvements are needed to solve larger examples in reasonable time. In our future work we expect to explore other practical optimizations, for instance by relaxing the relation of equivalence that we impose between the original rule system and its implementation. Another important research line will be to enrich our method with automatic or assisted ways to fix the detected problems.

References

1. Zacharias, V.: Development and verification of rule based systems - a survey of developers. In: Bassiliades, N., Governatori, G., Paschke, A. (eds.) RuleML 2008. LNCS, vol. 5321, pp. 6–16. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88808-6_4
2. Coenen, F., Eaglestone, B., Ridley, M.J.: Verification, validation, and integrity issues in expert and database systems: two perspectives. Int. J. Intell. Syst. **16**(3), 425–447 (2001)
3. Paschke, A.: Verification, validation and integrity of distributed and interchanged rule based policies and contracts in the semantic web. In: Semantic Web, 2nd International Semantic Web Policy Workshop (SWPW 2006). CEUR-WS.org (2006)
4. Han, W., Lei, C.: A survey on policy languages in network and security management. Comput. Netw. **56**(1), 477–489 (2012)
5. Hanamsagar, A., Jane, N., Borate, B., Wasvand, A., Darade, S.: Firewall anomaly management: a survey. Int. J. Comput. Appl. **105**(18), 1–5 (2014)
6. Aqib, M., Shaikh, R.A.: Analysis and comparison of access control policies validation mechanisms. I.J. Comput. Netw. Inf. Secur. **7**(1), 54–69 (2015)
7. Lin, D., Rao, P., Bertino, E., Li, N., Lobo, J.: Exam: a comprehensive environment for the analysis of access control policies. Int. J. Inf. Sec **9**(4), 253–273 (2010)
8. Hwang, J., Xie, T., Hu, V.C.: Detection of multiple-duty-related security leakage in access control policies. In: Secure Software Integration and Reliability Improvement, pp. 65–74. IEEE Computer Society (2009)
9. Montangero, C., Reiff-Marganiec, S., Semini, L.: Logic-based conflict detection for distributed policies. Fundamantae Informatica **89**(4), 511–538 (2008)
10. Halpern, J.Y., Weissman, V.: Using first-order logic to reason about policies. ACM Trans. Inf. Syst. Secur. **11**(4), 1–41 (2008)
11. Craven, R., Lobo, J., Ma, J., Russo, A., Lupu, E.C., Bandara, A.K.: Expressive policy analysis with enhanced system dynamicity. In: Li, W., Susilo, W., Tupakula, U.K., Safavi-Naini, R., Varadharajan, V. (eds.) Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, pp. 239–250. ACM (2009)
12. Turkmen, F., den Hartog, J., Ranise, S., Zannone, N.: Analysis of XACML policies with SMT. In: Focardi, R., Myers, A. (eds.) POST 2015. LNCS, vol. 9036, pp. 115–134. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46666-7_7

13. Ni, Q., et al.: Privacy-aware role-based access control. ACM Trans. Inf. Syst. Secur. **13**(3), 24:1–24:31 (2010)
14. Neri, M.A., Guarnieri, M., Magri, E., Mutti, S., Paraboschi, S.: Conflict detection in security policies using semantic web technology. In: Satellite Telecommunications (ESTEL), pp. 1–6. IEEE (2012)
15. Armando, A., Ranise, S.: Automated and efficient analysis of role-based access control with attributes. In: Cuppens-Boulahia, N., Cuppens, F., Garcia-Alfaro, J. (eds.) DBSec 2012. LNCS, vol. 7371, pp. 25–40. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31540-4_3
16. Hu, H., Ahn, G.J., Kulkarni, K.: Discovery and resolution of anomalies in web access control policies. IEEE Trans. Dependable Sec. Comput. **10**(6), 341–354 (2013)
17. Shaikh, R.A., Adi, K., Logrippo, L.: A data classification method for inconsistency and incompleteness detection in access control policy sets. Int. J. Inf. Sec. **16**(1), 91–113 (2017)
18. Deng, F., Zhang, L.Y.: Elimination of policy conflict to improve the PDP evaluation performance. J. Netw. Comput. Appl. **80**, 45–57 (2017)
19. Xia, X.: A conflict detection approach for XACML policies on hierarchical resources. In: Conference on Green Computing and Communications, Conference on Internet of Things, and Conference on Cyber, Physical and Social Computing, pp. 755–760. IEEE Computer Society (2012)
20. Royer, J.-C., Santana De Oliveira, A.: AAL and static conflict detection in policy. In: Foresti, S., Persiano, G. (eds.) CANS 2016. LNCS, vol. 10052, pp. 367–382. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48965-0_22
21. Liffiton, M.H., Malik, A.: Enumerating infeasibility: finding multiple MUSes quickly. In: Gomes, C., Sellmann, M. (eds.) CPAIOR 2013. LNCS, vol. 7874, pp. 160–175. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38171-3_11
22. Previti, A., Marques-Silva, J.: Partial MUS enumeration. In: 27th AAAI Conference on Artificial Intelligence, Bellevue, Washington, pp. 818–825. AAAI Press (2013)
23. Wu, H.: Finding achievable features and constraint conflicts for inconsistent metamodels. In: Anjorin, A., Espinoza, H. (eds.) ECMFA 2017. LNCS, vol. 10376, pp. 179–196. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61482-3_11
24. Adi, K., Bouzida, Y., Hattak, I., Logrippo, L., Mankovskii, S.: Typing for conflict detection in access control policies. In: Babin, G., Kropf, P., Weiss, M. (eds.) MCETECH 2009. LNBIP, vol. 26, pp. 212–226. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01187-0_17
25. Dunlop, N., Indulska, J., Raymond, K.: Methods for conflict resolution in policy-based management systems. In: Enterprise Distributed Object Computing Conference, pp. 98–111. IEEE Computer Society (2003)
26. 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). https://doi.org/10.1007/978-3-540-78800-3_24
27. Fisler, K., Krishnamurthi, S., Meyerovich, L.A., Tschantz, M.C.: Verification and change-impact analysis of access-control policies. In: International Conference on Software Engineering (2005)

Study of Integrating Random and Symbolic Testing for Object-Oriented Software

Marko Dimjašević[1], Falk Howar[2], Kasper Luckow[3], and Zvonimir Rakamarić[1](✉)

[1] School of Computing, University of Utah, Salt Lake City, UT, USA
{marko,zvonimir}@cs.utah.edu

[2] Department of Computer Science, Dortmund University of Technology, Dortmund, Germany
falk.howar@tu-dortmund.de

[3] Carnegie Mellon University Silicon Valley, Mountain View, CA, USA
kasper.luckow@sv.cmu.edu

Abstract. Testing is currently the main technique adopted by the industry for improving the quality, reliability, and security of software. In order to lower the cost of manual testing, automatic testing techniques have been devised, such as random and symbolic testing, with their respective trade-offs. For example, random testing excels at fast global exploration of software, while it plateaus when faced with hard-to-hit numerically-intensive execution paths. On the other hand, symbolic testing excels at exploring such paths, while it struggles when faced with complex heap class structures. In this paper, we describe an approach for automatic unit testing of object-oriented software that integrates the two techniques. We leverage feedback-directed unit testing to generate meaningful sequences of constructor+method invocations that create rich heap structures, and we in turn further explore these sequences using dynamic symbolic execution. We implement this approach in a tool called JDoop, which we augment with several parameters for fine-tuning its heuristics; such "knobs" allow for a detailed exploration of the various trade-offs that the proposed integration offers. Using JDoop, we perform an extensive empirical exploration of this space, and we describe lessons learned and guidelines for future research efforts in this area.

1 Introduction

The software industry nowadays heavily relies on testing for improving the quality of its products. There are, of course, good reasons for adopting this practice. First, as opposed to more heavy-weight techniques such as static analysis, testing is easy to deploy and understand, and most developers are familiar with software testing processes and tools. Second, testing is scalable (i.e., millions of tests can

Supported in part by the National Science Foundation (NSF) award CCF 1421678.

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 89–109, 2018.
https://doi.org/10.1007/978-3-319-98938-9_6

```
public class HardToHit {
  private int x;
  public HardToHit(int x) {
    this.x = (x < 0) ? -x : x;
  }
  public void setX(int x) {
    this.x = x;
  }
  public int distance(int y) {
    y = (y < 0) ? -y : y;
    int out;
    if (x > y) out = x - y;
    else out = y - x;
    assert out >= 0;
    return out;
  }
}
```

```
// Assertion Violation
public void testHardToHit() {
  HardToHit h = new HardToHit(0);
  h.setX(-1);
  h.distance(Integer.MAX_VALUE);
}
// Random Parameters
public void testHardToHit1() {
  HardToHit h =
      new HardToHit(random());
  h.distance(random());
}
// Random Method Sequence
public void testHardToHit2() {
  HardToHit h = new HardToHit(0);
  h.setX(0);
  h.distance(0);
}
```

Fig. 1. Class with an assertion in one method (left). Input `x` is not properly sanitized in method `setX`. Consequence: assertion can be violated by combination of method sequence and specific input values (right).

be executed within hours even on large programs) and precise (i.e., it does not generate false alarms that impede developers' productivity). Third, while testing cannot prove the absence of bugs, there is ample evidence that testing does find important bugs that are fixed by developers. Despite these advantages, testing is not a silver bullet since crafting good tests is a time consuming and costly process, and even then achieving high coverage and catching all defects using testing can be challenging. For example, tester to developer ratio at Microsoft is around 1-to-1, and yet important defects still escape into production. Naturally, there has been a great deal of research on alleviating these problems by developing techniques that aim to improve the automation and effectiveness (in terms of achieved coverage and defects found) of software testing.

Random testing is the most basic and straightforward approach to automating software testing. Typically, it completely automatically generates and executes millions of test cases within hours, and quickly covers many statements (or branches) of a *software under test* (SUT). However, a drawback of random testing is that, depending on the characteristics of the SUT, the achieved coverage plateaus due to unlikely execution paths. Figure 1 gives our motivating example Java program that illustrates this point (left) together with a specific test case that triggers an assertion violation (top right). To apply random testing on the example, we generate randomized unit test shown in the middle of the right half of the figure. Clearly, it is trivial to execute this simple unit test many times, each time with a new pair of random numbers being generated. It is impossible, however, that executing it would generate inputs that violate the assertion. We would additionally need to generate more complex sequences of method calls

(as is shown in lower right of the figure). Exploring both dimensions (parameter values and method sequences) randomly tends to plateau and not hit paths that require specifc combinations of method sequence and parameters values.

A more heavyweight approach could be based on symbolic execution, which leverages automatic constraint solvers to compute test inputs that cover such hard-to-cover branches. For example, the JDart [26] dynamic symbolic execution tool when run on method `testHardToHit2` generates test cases covering all branches in less than a second, thereby triggering an assertion violation. The authors also show that JDart improves coverage over random testing for a class of numerically-intensive SUTs. However, symbolic-testing-based methods mainly excel in automatically generating test inputs over primitive numeric data types, and have hence been successfully applied as either system-level (e.g., SAGE [18], KLEE [6]) or method-level (e.g., JDart [26], JCute [35]) test generators.

Generalizing from the above example, generating unit tests for object-oriented software poses a two-dimensional challenge: instead of taking just primitive types as input, methods in object-oriented software require a rich heap structure of class objects to be generated. While several approaches have been proposed that automatically generate symbolic heap structures [25], logical encoding of such structures results in more complex constraints that put an additional burden on constraint solvers; hence, these approaches have not yet seen wider adoption on large SUTs. On the other hand, generating heap structures by randomly creating sequences of constructor+method invocations was shown to be effective, in particular when advanced search- and feedback-directed algorithms are employed (e.g., Randoop [29], EvoSuite [13]). It is then natural to attempt to integrate the two approaches by using random testing to perform global/-macro exploration (by generating heap structures using sequences of constructor+method invocations at the level of classes) and dynamic symbolic execution to perform local/micro exploration (by generating inputs of primitive types using constraint solvers at the level of methods). In this paper, we describe, implement, and empirically evaluate such a hybrid approach.

Our hybrid approach integrates feedback-directed unit testing with dynamic symbolic execution. We leverage feedback-directed unit testing to generate constructor+method sequences that create heap structures and drive a SUT into interesting global (i.e., macro) states. We feed the generated sequences to a dynamic symbolic execution engine to compute inputs of primitive types that drive the SUT into interesting local (i.e., micro) states. We implemented this approach as a tool named JDoop,[1] which integrates feedback-directed unit testing tool Randoop [29] with state-of-the-art dynamic symbolic execution engine JDart [26]. Given that such an integration has not been thoroughly empirically studied in the past, we also assess the merits of this approach through a large-scale empirical evaluation.

[1] Note that a very preliminary version of JDoop was presented earlier as a short workshop extended abstract [11].

Our main contributions are as follows:

- We developed JDoop, a hybrid tool that integrates feedback-directed unit testing with dynamic symbolic execution to be able to experiment with large-scale automatic testing of object-oriented software.
- We implemented a distributed benchmarking infrastructure for running experiments in isolation on a cluster of machines; this allows us to execute large-scale experiments that ensure statistical significance, and also advances the reproducibility of our results.
- We performed an extensive empirical evaluation and comparison between random (our baseline) and hybrid testing approaches in the context of automatic testing of object-oriented software.
- We identified several open research questions during our evaluation, performed additional targeted experiments to obtain answers to these questions, and provided guidelines for future research efforts in this area.

2 Background

We provide background on dynamic symbolic execution and feedback-directed random testing.

2.1 Dynamic Symbolic Execution

Dynamic symbolic execution [6,17,36] is a program analysis technique that executes a program with concrete and symbolic inputs at the *same* time. It systematically collects constraints over the *symbolic* program inputs as it is exploring program paths, thereby representing program behaviors as algebraic expressions over symbolic values. The program effects can thus be expressed as a function of such expressions.

Dynamic symbolic execution maintains—in addition to the concrete state defined by the concrete program semantics—the symbolic state, which is a tuple containing symbolic values of program variables, a path condition, and a program counter. A path condition is a conjunction of symbolic expressions over the symbolic inputs that characterizes an execution path through the program. It is generated by accumulating (symbolic) conditions encountered along the execution path, so that concrete data values that satisfy it can be used to drive its concrete execution. Path conditions are stored as a *symbolic execution tree* that characterizes all the paths exercised as part of the symbolic analysis.

In dynamic symbolic execution, the symbolic execution tree is built by repeatedly augmenting it with new paths that are obtained from unexplored branches in the tree. This is done by employing an exploration strategy such as depth-first, breadth-first, or random. A constraint solver is used to obtain a valuation for a yet-unexplored branch by feeding it the corresponding path condition. The new valuation drives a new iteration of dynamic symbolic execution that augments the symbolic execution tree with a new path. JDart is a dynamic

symbolic execution engine that uses the Java PathFinder framework [23,44] and for executing Java programs and recording path conditions. Maintaining the symbolic state is achieved by a customized implementation of the bytecode instructions in the JVM of Java PathFinder that performs concrete and symbolic operations simultaneously. In JDoop, we configure JDart to use the Z3 [9] constraint solver for finding concrete inputs that drive execution along previously unexplored symbolic paths.

A limitation of this approach is that native code is outside the scope of the analysis. Based on the Nhandler extension [38] to Java PathFinder, JDart offers two strategies for dealing with native code.

- **Concrete Native.** In this mode, JDart executes native code on concrete data values, and no symbolic execution of native parts is performed—only concrete values are passed to and from native calls, and symbolic values are not updated and cannot taint native return values. The return value is annotated with a new symbolic variable. As a consequence, the concrete side of an execution is faithful to the respective execution on a normal JVM. However, branches in the native code are not recorded in symbolic path conditions, which can lead to JDart not being able to explore branches after a native call as well. Another downside of this mode is that the implementation in Java PathFinder is relatively slow.
- **No Native.** In this mode, JDart does not execute native code at all. Instead, it returns a default concrete value every time a native method is called and a return value is expected. The concrete value is annotated with the corresponding symbolic variable, using the method signature of the native method as the name of that variable. Concrete execution, in this case, is not faithful to the respective execution on a normal JVM as the introduced default values in most cases are not equal to the values that would be returned by the actual method invocations (and side effects are ignored as well). Recorded symbolic branches cannot be explored even if solutions are found by a constraint solver as there currently is no mechanism that allows feeding these values into the execution (instead of the default return values of native methods).

Since the 'No Native' mode is more performant and since currently there is no way of solving most of the recorded constraints in 'Concrete Native' mode (cf. results in Sect. 4), JDoop runs JDart in 'No Native' mode for native code. We use the 'Concrete Native' mode in our evaluation for analyzing the potential limiting impact of not executing native code faithfully and not being able to find and inject values that target branches in native code.

JDart produces the following outputs: a symbolic execution tree that contains all explored paths along with performance statistics, vectors of concrete input values that execute paths in the tree, and a suite of test cases (based on these vectors). A symbolic execution tree contains leaf nodes for all explored paths and additionally leaves for branches off of executed paths that could not be explored because the constraint solver was not able to produce adequate concrete values or because native code is not executed (in fully symbolic mode). For these leaves JDart does not generate input vectors or test cases.

2.2 Feedback-Directed Random Testing

A simple approach to automatic unit testing of object-oriented software is to completely randomly generate sequences of constructor+method invocations together with the respective concrete input values. However, this typically results in a large overhead since numerous sequences get generated with invalid prefixes that lead to violations of common implicit class or method requirements (e.g., passing null reference to a method that expects an allocated object). Moreover, such sequences cause trivial, uninteresting exceptions to be thrown early, thereby preventing deep exploration of the SUT state space. Hence, instead of generating unit tests blindly and in a completely random fashion, useful feedback can be gathered from previous test executions to direct the creation of new unit tests. In this way, unit tests that execute long sequences of method calls to completion (i.e., without exceptions being thrown) can be generated. This approach is known as *feedback-directed random testing* and is implemented in the Randoop automatic unit testing tool [29].

Randoop uses information from previous test executions to direct further unit test generation. The tool maintains two sets of constructor+method invocation sequences: those that do not violate a property (i.e., property-preserving) and those that do (i.e., property-violating). The property-violating set is initially empty, while the property-preserving set is initialized with an empty sequence. The default property that is maintained is unit test termination without any errors or exceptions being thrown. Randoop randomly selects a public method (or a constructor) and an existing sequence from the property-preserving set. It then appends the invocation of the selected constructor/method to the end of the sequence, and replaces primitive type arguments with concrete values that are randomly selected from a preset pool of values. Next, the newly generated sequences are compared against all previously generated sequences in the two sets. If it already exists, it is simply dropped and random selection is repeated. Otherwise, Randoop executes the new sequence and checks for property violations. If no properties are violated, the sequence is added to the property-preserving set and otherwise to the property-violating set. Randoop keeps on extending property-preserving sequences until it reaches a provided time limit.

3 Hybrid Approach

In this section, we describe our hybrid approach that integrates dynamic symbolic execution and feedback-directed random testing into an algorithm for automatic testing of object-oriented software. We implemented this algorithm as the JDoop tool that is freely available.[2] Figure 2 shows the flow of the algorithm, which is iterative and each iteration consists of several stages that we describe next.

[2] JDoop is available under the GNU General Public License version 3 (or later) at https://github.com/psycopaths/jdoop.

3.1 Generation of Sequences

The first stage of every iteration of our algorithm is feedback-directed random testing using Randoop, which generates constructor+method sequences as described in Sect. 2.2. Randoop takes advantage of a pool of concrete primitive values to be used as constructor/method arguments when generating sequences. In the first iteration, we use the default pool with few values, which for the integer type are −1, 0, 1, 10, 100. Hence, an instance of a generated sequence for our running example from Fig. 1 is the one shown in the middle of the right half of the figure. Our algorithm grows the pool for subsequent iterations with concrete inputs generated by dynamic symbolic execution, which we describe later. The sequences generated in this stage serve two purposes. First, we employ them as standalone unit tests that exercise the SUT, which is their original intended purpose. Second, our hybrid algorithm also employs them as *driver programs* to be used in the subsequent dynamic symbolic execution stage.

3.2 Selection and Transformation of Sequences

The previous stage typically generates far too many sequences to be successfully explored with a dynamic symbolic execution engine in a reasonable amount of time. For example, several thousands of valid sequences are often generated in just a few seconds. Hence, it is prudent to select a promising subset of the generated sequences to be transformed into inputs for the subsequent dynamic symbolic execution with JDart. The second stage implements the selection and transformation of constructor+method sequences.

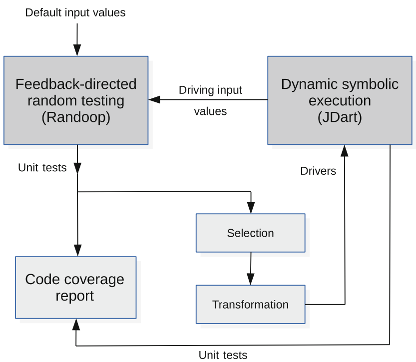


Fig. 2. Iterative algorithm of JDoop for unit test generation. The algorithm integrates dynamic symbolic execution and feedback-directed random testing.

Note that dynamic symbolic execution techniques have limitations, which is why we implemented the hybrid approach in the first place. In particular, they can typically treat symbolically only method arguments of primitive types. For example, if a sequence contains method calls with non-primitive types only, JDart will not be able to explore any additional paths. Hence, not every generated sequence is suitable for dynamic symbolic execution with JDart, and as the first step of this stage, we filter out all sequences with no arguments of a primitive type. Next, we have two strategies (i.e., heuristics) for selecting promising sequences. The first strategy randomly selects a subset of sequences. The second strategy prioritizes candidate sequences with more symbolic variables, which is based on the intuition that having more symbolic variables leads to more paths (and also branches and instructions) being covered. We compare the two strategies in our empirical evaluation. Once promising sequences are selected, they have to be appropriately transformed into driver programs for JDart.

Every candidate sequence is transformed for the final stage that performs dynamic symbolic execution. We achieve this by turning all constructor and method arguments of primitive types, which are supported by JDart, into symbolic input values. In our implementation, this is a simple source-to-source transformation. For instance, our example sequence results in the following driver program:

```
public class TestClass {
  void test1(int s0,int s1,int s2) {
    HardToHit h = new HardToHit(s0);
    h.setX(s1);
    h.distance(s2);
  }
  static void main(String[] a) {
    TestClass tc = new TestClass();
    @Symbolic int x, y, z;
    tc.test1(x, y, z);
  }
}
```

In the driver, the integer inputs to constructor `HardToHit` and methods `setX` and `distance` are transformed into the arguments of the `test1` test method. The `test1` method is called from the main method that is added as an entry point for dynamic symbolic execution. Finally, JDart is instructed that the `s0`, `s1`, and `s2` inputs to `test1` are treated symbolically.

3.3 Dynamic Symbolic Execution of Sequences

The last stage of every iteration is exploring the generated driver programs using dynamic symbolic execution as implemented in JDart. JDart explores paths through each driver program by solving path constraints over the specified symbolic inputs as described in Sect. 2.1. In the process, it generates additional unit tests, where each unit test corresponds to an explored path. The generated unit tests are added into the final set of unit tests. In addition to generating these unit tests, we also collect all the concrete input values that JDart generates in the process. We add these values back into the Randoop's concrete primitive value pool for the sequence generation stage of the next iteration. By doing this, we feed the information that the dynamic symbolic execution generates back into the feedback-directed random testing stage.

4 Empirical Evaluation

We aim to answer the following research questions using the results of our empirical evaluation.

1. Can JDoop cover paths that plain random test case generation does not, and how big is the positive impact of covering such paths? To answer this question, we compare the performance of Randoop (as our baseline) and JDoop, using code coverage as a metric for the quality of the generated test suites.

Table 1. SF110 Benchmarks we use in the evaluation. Column #B is the number of branches, #I instructions, #M methods, and #C classes.

Benchmark	#B	#I	#M	#C	Benchmark	#B	#I	#M	#C
1_tullibee	915	8402	204	19	47_dvd-homevideo	376	10670	161	48
2_a4j	544	9773	522	45	48_resources4j	312	3223	104	12
3_gaj	22	415	52	10	49_diebierse	197	4859	185	19
5_templateit	564	5391	195	23	50_biff	814	7348	49	6
6_jnfe	132	7545	339	52	53_shp2kml	26	656	30	6
7_sfmis	146	4386	185	19	55_lavalamp	128	2907	236	48
9_falselight	16	1189	32	14	63_objectexplorer	959	14118	902	84
11_imsmart	103	2244	86	17	65_gsftp	517	6587	181	32
13_jdbacl	3098	49385	1578	198	67_gae-app-manager	68	1405	46	8
14_omjstate	52	954	67	14	68_biblestudy	424	6005	313	23
16_templatedetails	38	656	87	24	69_lhamacaw	2016	51698	1437	101
22_byuic	2124	15031	195	14	72_battlecry	674	9550	130	15
23_jwbf	949	16032	609	86	74_fixsuite	374	6520	241	36
26_jipa	128	1488	36	5	76_dash-framework	12	188	37	17
28_greencow	0	7	2	1	79_twfbplayer	1132	18315	902	160
30_bpmail	208	3372	208	32	84_ifx-framework	299	136363	26257	3900
31_xisemele	150	3036	269	50	90_dcparseargs	88	654	21	6
34_sbmlreader2	76	1447	26	8	94_jclo	110	1094	43	4
37_petsoar	208	3445	377	58	95_celwars2009	850	15208	164	32
42_asphodel	64	1139	101	20	98_trans-locator	40	1097	39	6
46_nutzenportfolio	1183	18335	826	62					

2. Can dynamic symbolic execution enable randomized test case generation to access regions of a SUT that remain untested otherwise, i.e., does the feedback loop from JDart to Randoop (see Fig. 2) have a measurable impact on achieved coverage? To answer this question, we run JDoop in multiple configurations with varying amounts of runtime attributed to Randoop and JDart, enabling a feedback loop in some configurations and preventing it in others.
3. What are the constituting factors impacting the effectiveness of JDoop in terms of the code coverage that can be achieved through automated generation of test suites? More specifically, can we confirm or refute the conjecture from related work [14] that robustness of the used dynamic symbolic execution engine is pivotal or do other factors exist that have an impact on the achievable coverage (e.g., selection of test cases for symbolic execution)? To answer this question, we analyze statistics produced by JDart and vary the strategy in JDoop for selecting method sequences for execution with JDart as discussed in Sect. 3 (either selecting sequences randomly or prioritizing those with many symbolic variables).

In the remainder of this section, we introduce the benchmarks we used in our evaluation, describe our experimental setup, and present and discuss the results of the evaluation.

4.1 Benchmarks

We performed our empirical evaluation using the SF110 benchmark suite [37]. The suite consists of 110 Java projects that were randomly selected from the

SourceForge repository of free software to reduce the threat to external validity (see Sect. 5). In our evaluation, we chose the largest subset of SF110 that both JDoop and Randoop can successfully execute on. Benchmarks that were excluded can be grouped into the following categories: unsuitable environment, inadequate or empty benchmarks, and deficiencies of testing tools. In the unsuitable environment category, benchmarks require privileged permissions in the operating system, a properly set configuration file, or a graphical subsystem to be available. There are several empty benchmarks, benchmarks that call the `System.exit()` method that is not trapped by testing tools, and benchmarks that are otherwise inadequate because of conflicting dependencies with our testing infrastructure. Finally, for some benchmarks Randoop generates test cases that do not compile. All such problematic benchmarks were excluded from consideration, which left us with 41 benchmarks total, as listed in Table 1. For each benchmark we list the number of instructions, branches, methods, and classes, which demonstrates we use a wide range of SUTs in terms of their size and complexity.

4.2 Experimental Setup

We used two tools in our empirical evaluation: JDoop and Randoop (version 3.0.10). We explored several configurations of JDoop, where each configuration is determined by three parameters. The first parameter is the time limit for the first stage of every iteration, which is when Randoop runs (see Sect. 2.2); we vary this parameter as 1, 9, and 20 min. The second parameter is the time limit for the second and third stages combined, which is when JDart runs; we vary this parameter as 1, 9, and 40 min. The third parameter determines the strategy for selecting constructor+method call sequences as candidates for dynamic symbolic execution between: (1) random selection (denoted by R), and (2) prioritization based on the number of symbolic variables (denoted by P). Each configuration is code-named as JD-O-J-S, where O is the time limit for Randoop, J is the time limit for JDart, and S is the sequence selection strategy used. We explored the following six JDoop configurations: JD-1-9-P, JD-1-9-R, JD-9-1-P, JD-9-1-R, JD-20-40-P, and JD-20-40-R.

We carried out the evaluation in the Emulab testbed infrastructure [45]. We used 20 identical machines, each of which was equipped with two 2.4 GHz 64-bit 8-core processors, 64 GB of DDR4 RAM, and an SSD disk; the machines were running Ubuntu 16.04. We developed our testing infrastructure around the Apache Spark cluster computing framework. To facilitate reproducibility, each execution of a testing tool on a benchmark is performed in a pristine sandboxed virtualization environment. This is achieved via LXC containers running a reproducible build of Debian GNU/Linux code-named Stretch. We allocated 4 dedicated CPU cores and 8 GB of RAM to each container. Both Randoop and JDoop are multi-threaded, and hence they utilized the multiple available CPU cores. Our testing infrastructure is freely available for others to use and extend.[3]

[3] The testing infrastructure is available under the GNU Affero GPLv3+ license at https://github.com/soarlab/jdoop-wrapper.

Table 2. Branch coverage (including standard deviations) averaged across 5 runs. The highest and lowest numbers per benchmark are given in bold and italic, respectively.

Benchmark	Randoop	JD-1-9-P	JD-1-9-R	JD-9-1-P	JD-9-1-R	JD-20-40-P	JD-20-40-R
1_tullibee	*28.0* ± 1.2	29.4 ± 1.4	29.7 ± 1.0	29.4 ± 0.0	**31.6 ± 0.5**	28.7 ± 0.0	29.0 ± 0.2
2_a4j	58.5 ± 0.5	*57.9 ± 0.0*	60.0 ± 0.6	59.6 ± 0.2	**62.4 ± 0.1**	60.7 ± 0.1	60.7 ± 0.0
3_gaj	40.9 ± 0.0	40.9 ± 0.0	40.9 ± 0.0	40.9 ± 0.0	40.9 ± 0.0	40.9 ± 0.0	40.9 ± 0.0
5_templateit	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0
6_jnfe	48.5 ± 0.0	48.5 ± 0.0	48.5 ± 0.0	48.5 ± 0.0	48.5 ± 0.0	48.5 ± 0.0	48.5 ± 0.0
7_sfmis	*35.9 ± 0.9*	40.4 ± 4.2	39.7 ± 4.2	**42.5 ± 0.0**	40.5 ± 5.5	37.5 ± 2.7	37.1 ± 2.7
9_falselight	6.3 ± 0.0	6.3 ± 0.0	6.3 ± 0.0	6.3 ± 0.0	6.3 ± 0.0	6.3 ± 0.0	6.3 ± 0.0
11_imsmart	17.5 ± 0.0	17.5 ± 0.0	17.5 ± 0.0	17.5 ± 0.0	17.5 ± 0.0	17.5 ± 0.0	17.5 ± 0.0
13_jdbacl	36.6 ± 0.7	32.2 ± 3.1	32.2 ± 1.8	37.0 ± 0.5	**38.5 ± 0.6**	34.2 ± 1.0	33.6 ± 0.8
14_omjstate	48.1 ± 0.0	48.1 ± 0.0	48.1 ± 0.0	48.1 ± 0.0	**48.8 ± 3.1**	42.3 ± 0.0	42.3 ± 0.0
16_templatedetails	71.1 ± 0.0	68.4 ± 0.0	70.0 ± 1.8	71.1 ± 0.0	71.1 ± 0.0	68.4 ± 0.0	68.4 ± 0.0
22_byuic	7.8 ± 0.0	7.8 ± 0.0	7.8 ± 0.0	7.8 ± 0.0	7.8 ± 0.0	7.8 ± 0.0	7.8 ± 0.0
23_jwbf	26.6 ± 2.1	26.5 ± 1.7	27.2 ± 0.9	28.0 ± 0.6	**28.2 ± 1.9**	26.1 ± 0.5	*26.0 ± 0.0*
26_jipa	*18.8 ± 0.0*	24.2 ± 0.0	24.2 ± 0.0	24.2 ± 0.0	24.2 ± 0.0	23.4 ± 0.0	23.4 ± 0.0
28_greencow	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
30_bpmail	36.9 ± 0.5	36.9 ± 1.5	*36.1 ± 1.2*	**37.3 ± 0.6**	37.2 ± 0.6	37.2 ± 0.6	37.1 ± 0.5
31_xisemele	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
34_sbmlreader2	10.5 ± 0.0	10.5 ± 0.0	10.5 ± 0.0	10.5 ± 0.0	10.5 ± 0.0	10.5 ± 0.0	10.5 ± 0.0
37_petsoar	**54.1 ± 0.7**	*52.8 ± 1.6*	52.9 ± 1.4	53.4 ± 0.0	53.4 ± 0.0	53.7 ± 0.7	53.7 ± 0.7
42_asphodel	9.4 ± 0.0	9.4 ± 0.0	9.4 ± 0.0	9.4 ± 0.0	9.4 ± 0.0	9.4 ± 0.0	9.4 ± 0.0
46_nutzenportfolio	5.5 ± 0.0	*5.2 ± 0.0*	5.3 ± 1.6	5.6 ± 0.0	5.6 ± 0.6	5.5 ± 0.0	5.5 ± 0.0
47_dvd-homevideo	0.8 ± 0.0	0.8 ± 0.0	0.8 ± 0.0	0.8 ± 0.0	0.8 ± 0.0	0.8 ± 0.0	0.8 ± 0.0
48_resources4j	0.6 ± 0.0	0.6 ± 0.0	0.6 ± 0.0	0.6 ± 0.0	0.6 ± 0.0	0.6 ± 0.0	0.6 ± 0.0
49_diebierse	13.7 ± 0.0	*13.4 ± 3.0*	**19.7 ± 1.0**	14.2 ± 0.0	15.2 ± 15.1	13.7 ± 0.0	18.6 ± 13.1
50_biff	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0
53_shp2kml	19.2 ± 0.0	19.2 ± 0.0	19.2 ± 0.0	19.2 ± 0.0	19.2 ± 0.0	19.2 ± 0.0	19.2 ± 0.0
55_lavalamp	49.8 ± 0.6	48.4 ± 0.0	48.8 ± 1.6	51.9 ± 0.7	**52.0 ± 0.7**	48.4 ± 0.0	*48.0 ± 2.0*
63_objectexplorer	25.3 ± 0.0	24.6 ± 1.8	*24.5 ± 1.0*	**26.4 ± 0.3**	26.3 ± 0.9	25.0 ± 0.0	25.0 ± 0.2
65_gsftp	9.8 ± 1.0	9.9 ± 1.0	**10.0 ± 0.9**	9.9 ± 0.0	9.9 ± 0.0	9.5 ± 0.0	9.5 ± 0.0
67_gae-app-manager	2.9 ± 0.0	2.9 ± 0.0	2.9 ± 0.0	2.9 ± 0.0	2.9 ± 0.0	2.9 ± 0.0	2.9 ± 0.0
68_biblestudy	**37.5 ± 0.0**	*36.9 ± 0.7*	37.0 ± 0.4	37.3 ± 0.0	37.2 ± 0.8	37.0 ± 0.0	37.0 ± 0.0
69_lhamacaw	42.7 ± 0.4	40.1 ± 0.6	*39.9 ± 1.0*	**46.1 ± 0.5**	45.7 ± 0.6	40.3 ± 0.7	40.1 ± 0.4
72_battlecry	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0
74_fixsuite	17.5 ± 6.5	17.1 ± 3.1	*15.5 ± 1.3*	19.2 ± 1.8	**19.6 ± 4.0**	17.4 ± 2.8	17.2 ± 3.3
76_dash-framework	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0
79_twfbplayer	27.3 ± 0.0	23.2 ± 1.8	*21.5 ± 1.3*	29.4 ± 0.0	29.3 ± 1.0	**29.5 ± 0.0**	29.4 ± 0.1
84_ifx-framework	30.8 ± 0.0	32.6 ± 9.7	31.0 ± 8.9	**32.9 ± 2.8**	32.0 ± 2.8	29.5 ± 4.9	*28.8 ± 2.3*
90_dcparseargs	64.8 ± 0.0	64.8 ± 0.0	64.8 ± 0.0	64.8 ± 0.0	64.8 ± 0.0	64.8 ± 0.0	64.8 ± 0.0
94_jclo	*42.7 ± 0.0*	**46.0 ± 1.6**	44.5 ± 0.0	44.5 ± 0.0	44.7 ± 0.8	44.5 ± 0.0	44.5 ± 0.0
95_celwars2009	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0	**2.2 ± 5.2**	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0
98_trans-locator	25.0 ± 0.0	*15.0 ± 36.5*	18.0 ± 37.7	25.0 ± 0.0	**27.0 ± 3.7**	25.0 ± 0.0	25.0 ± 0.0

We allocate a one hour time limit per benchmark per testing tool/configuration for test case generation. Subsequent test case compilation and code coverage measurement phases are not counted toward the 1 h time limit. Given that both Randoop and JDoop employ randomized heuristics, we repeat each run 5 times to account for this variability—for each benchmark we compute an average and a standard deviation. In terms of code coverage metrics, we measured instruction and branch coverage at the Java bytecode level using JaCoCo [20]. Furthermore, to get more insight into the performance of JDart, we collect statistics on the number of successful and failed runs, additional test cases it generates, symbolic variables in driver programs, times a constraint solver could not find valuation for a path condition, and JDart runs that explored one path versus multiple paths.

4.3 Evaluation of Test Coverage

Table 2 gives branch coverage results for each tool and configuration on all of the benchmarks. Most results are stable across multiple runs, meaning that the calculated standard deviations are very small. In particular, the standard deviations for Randoop on a vast majority of benchmarks are 0, even though we used a different random seed for every run. This suggests that Randoop reaches

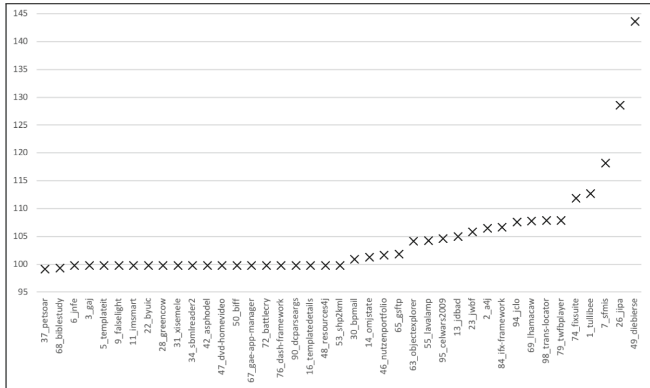


Fig. 3. Increases in branch coverage per benchmark by JDoop over baseline of Randoop (in % of coverage by baseline).

saturation and is unable to cover more branches. For the most part there are only small differences in the achieved coverage between different tools/configurations when looking at the total number of covered branches. However, JDoop (in one of its configurations) consistently achieves higher coverage than Randoop. Given that pure Randoop saturates, we can conclude that the improvements in coverage we observe with JDoop are due to leveraging dynamic symbolic execution. Among JDoop configurations, best-performing are the two 9-1 configurations where in an iteration Randoop runs for 9 min and JDart for 1 min; there are 6 such iterations in the 1 h time limit.

Figure 3 shows the increase in branch coverage per benchmark over pure Randoop that is achieved by some configuration of JDoop. The increase is measured as a percent increase in number of branches covered by JDoop over Randoop. Standard deviation is omitted in this graph as it was small in most instances (cf. Table 2). In two benchmarks JDoop performes slightly worse than pure Randoop. In roughly half of the benchmarks no change is observed—and in most cases with no variance. This suggests that these benchmarks are simply not amenable to increasing coverage by use of symbolic execution. In the remaining half of the benchmarks, branch coverage is increased. Increases range from 101.1% to 143.8% achieved coverage relative to the baseline of pure Randoop, with an average increase of 109.6% across this half of benchmarks.

4.4 Profiling Dynamic Symbolic Execution

To analyze the potential impact of the robustness of dynamic symbolic execution on the validity of our results, we collected data from runs on all benchmarks for all configurations. We perform this analysis on data from single runs of JDoop as the other results show very little variation of results between different runs in most cases. Table 3 reports statistics on the JDart operation in different series of experiments. Data in the table is explained and discussed in the following paragraphs.

Table 3. Statistics produced by JDART for single runs of all benchmarks in different configurations of JDOOP. JDART uses NHANDLER in the 'No Native' mode, except for one experiment that we performed in the 'Concrete Native' (CN) mode.

Sequence Selection Strategy	JD-20-40 R	JD-20-40 P	JD-1-9 R	JD-1-9 P	JD-9-1 R	JD-9-1 P	JD-9-1 (CN) R
Potential Impact / Best Mode of Operation							
# Successful Runs	33,390	28,316	46,976	43,770	4,629	1,017	3,885
Successful Runs (%)	98.5	97.8	98.2	97.5	98.5	100.0	96.3
# Additional Tests	6,436	10,802	11,272	16,588	914	5,382	n/a
# Benchmarks with Additional Tests	19	9	20	13	18	4	n/a
Robustness and Scalability of JDART							
# Failed Runs	507	648	853	1,121	69	0	148
due to unhandled native code	3	1	14	6	1	0	10
due to classloading in SUT	504	647	839	1,115	68	0	138
Failed Runs (%)	1.5	2.2	1.8	2.5	1.5	0.0	3.7
# D/K Paths	17	192	170	84	5	0	26,915
D/K Paths (%)	0.3	1.8	1.5	0.5	0.0	0.0	93.6
Amenable Test Cases							
# Symbolic Variables per Test Case (Avg.)	2.1	6.6	1.9	4.7	2.0	6.2	1.9
# Runs of Single Paths	32,410	27,293	45,268	42,162	4,495	988	2,801
# Runs with Multiple Paths	980	1,023	1,708	1,608	134	29	1,084

Modes of Operation. For all of the analyzed configurations of JDOOP, JDART runs successfully in the vast majority of cases and produces significant numbers of test cases (up to $16,588$ in total for all benchmarks in one experiment). Most additional test cases are produced in the JD-1-9 configurations that enable the feedback loop between RANDOOP and JDART but grant the bulk of runtime to JDART. Across all configurations, random selection of method sequences for JDART leads to generating additional test cases for more benchmarks than prioritizing sequences with many symbolic variables. Prioritization, on the other hand, leads to more additional test cases in total.

Robustness and Scalability. Our data indicates that JDART is robust. Only a small number of runs fail (between 0.0% and 2.5%). Of these failures, only a tiny fraction is due to unhandled native code (less than 1%).[4] The vast majority of failed runs is caused by class-path issues in the benchmarks (more than 99%). There are only very few cases in which the constraint solver was not able to solve constraints of all paths in symbolic execution trees (between 0.0% and 1.8%).

Using NHANDLER in the 'Concrete Native' mode leads to native calls being executed faithfully and to longer recorded path conditions, as discussed in Sect. 2. This yields constraints that are marked as not solvable ('don't know' or D/K for short) in 93.6% of all discovered paths in symbolic execution trees. This indicates the likelihood of JDART not being able to explore most of the paths that could be explored with proper symbolic treatment of native methods. Table 4 reports the

[4] These are methods for which NHANDLER was not configured to take over execution, leading to a crash of JDART. We configured NHANDLER to take care of all native methods of java.lang.String.

Table 4. Symbolic Variables introduced by NHANDLER in the 'Concrete Native' mode in a single run of JD-9-1.

Method	Occurrences
java.lang.String.charAt(I)C	2,157,258
java.lang.String.indexOf(I)I	430,951
java.lang.String.indexOf(II)I	18,199
java.lang.Character.isWhitespace(C)Z	63,723
java.lang.Character.isLetterOrDigit(C)Z	18,517
java.lang.Character.toLowerCase(C)C	16,506
java.lang.Math.min(II)I	2,800
java.lang.Float.floatToRawIntBits(F)I	81
sun.misc.Unsafe.compareAndSwapInt(Ljava/lang/Object;JII)Z	4,008

number of occurrences for all encountered native methods in one run of JDOOP. As can be seen from the data, the *charAt* method of the String class offers by far the greatest potential for improving on the number of explored paths. Note, however, that numbers in the table do not necessarily translate into the same number of additional paths as occurrences are counted along paths in trees and the same method call may appear on multiple paths.

Amenable Test Cases. The number of symbolic variables per test case behaves as expected: it increases when using prioritization of sequences with many variables. Prioritization, however, comes at a cost since there tends to be more runs of JDART in configurations that do not use prioritization. For all benchmarks, a high number of runs yields only one path and hence no additional test cases. A considerable number of these runs may be attributed to using NHANDLER in the 'No Native' mode, thereby hiding branches by not executing native code. On the other hand, even in the experiment in which NHANDLER was used in the 'Concrete Native' mode, two thirds of all runs explored only a single path. This indicates that many method sequences that were selected for JDART simply do not branch on symbolic variables.

4.5 Discussion

The obtained results allow us to provide answers to our research questions.

Question 1: Covering More Paths. JDOOP consistently outperforms RANDOOP on roughly 50% of the benchmarks (see Table 2 and Fig. 3). Measured in absolute number of branches, the margins are relatively slim in many cases. There are, however, cases in which the achieved branch coverage is increased by 28%—resulting in an increase in code coverage by 5.4% points (26_jipa). On about 50% of the benchmarks no variation can be seen in coverage between both approaches. Together with the little variance that is observed between different runs this indicates that RANDOOP in many cases reaches a state where achievable coverage is (nearly) saturated. It makes sense that in such a scenario JDOOP does not add many percentage points in code coverage. It merely adds coverage through those hard-to-hit corner cases.

Question 2: Reachable Regions. Our results indicate that the feedback loop has a positive impact. The JD-9-1 configurations perform better than other configurations in most cases. Regarding the time distribution between Randoop and JDart the picture is not as clear. There is a lot more variation in the margins of coverage increase (or decrease sometimes) for the configuration that grants most of the time to JDart. In one particularly amenable case this results in coverage increase by 43% (from 13.7% to 19.7% for 49_diebierse).

Question 3: Robustness of Symbolic Execution. Here, we have to refute the conjecture that was made in related work [14], namely that a robust dynamic symbolic execution engine can reap big increases in code coverage—or at least curb expectations about achievable coverage increases. Our experiments showed that JDart handles most benchmarks without many problems. Proper analysis of native code (especially for String methods) certainly has the potential to improve code coverage further, but the consistently high number of symbolic analyses that result in a single path (even in the control experiment) points to another important factor that contributes to small margins: the generated test cases simply do not allow exploring many new branches in most cases.

The experiments even indicate that it does not pay off to prioritize method sequences with many variables for JDart. Prioritization adds cost twice: once for analyzing test cases and then for exploring with many variables. Taking into account the observation from the first answer, that Randoop (almost) achieves saturation of coverage in one hour, this again indicates that in JDoop corner cases are discovered by JDart. Covering more search space beats investigating the few locations more intensively in such a scenario.

Remark on Achievable Coverage. Our observations correlate well with the observations made in [12], where the results of a static analysis of the SF110 benchmark suite are reported. The analysis revealed that only 6.6% of methods in the benchmark suite have path constraints that are exclusively composed of primitive type elements. On the other hand, the study identified objects in path constraints, calls to external libraries or native code, and exception-dependent paths as challenges to symbolic execution. The authors report that one third of methods have paths that deal with exceptions.

The low coverage (in absolute numbers) and low variance across all benchmarks for Randoop and JDoop in our experiments suggests that many branches simply cannot be covered by test cases that only rely on calling methods of objects from a project under test. Many branches rely on return values of calls to external libraries or the occurence of exceptions, which are not triggered in a simple testing environment. Since there is no simple or automated approach for determining the achievable coverage for a benchmark, we sampled a few individual benchmarks and indeed quickly found cases where `catch`-blocks in the code contained comments to the effect that the block is unreachable.

Taking into account the results from [12] and our findings, we conjecture that the branch coverage that is achieved by JDoop is close to the coverage that can be achieved without making the environment of a tested project symbolic.

5 Threats to Validity

Threats to External Validity. While the main purpose of the SF110 corpus of benchmarks is to reduce the threat to external validity since they were chosen randomly, we cannot be absolutely sure that the benchmarks we used are representative of Java programs. In addition, we excluded a number of problematic benchmarks from our evaluation (see Sect. 4.1). Hence, our results might not generalize to all programs. In JDoop we integrated Randoop and JDart, and we used Randoop as the baseline in our evaluation. We attempted to include another contemporary state-of-the-art Java testing tool into the comparison, and EvoSuite was an obvious choice to try. However, to the best of our ability we did not manage to get it to work with JaCoCo (the tool we use for measuring code coverage) on our benchmark suite despite exchanging numerous emails with the EvoSuite authors. This is a well-known problem caused by the online bytecode modifications that EvoSuite often performs.[5] While others successfully combined EvoSuite and JaCoCo in the past, that was accomplished only on very simple programs; in addition, others also reported differences in coverage results between EvoSuite's internal measurements and JaCoCo.[6,7] Hence, we could not perform a direct comparison and our results might not generalize to other tools. However, earlier work on EvoSuite reports similar results to ours with respect to using dynamic symbolic execution in combination with random testing [14]. Finally, note that we do not include the environment and dependencies of benchmarks into unit test generation, which might lead to sub-optimal coverage.

Threats to Internal Validity. In our evaluation, we experimented with 3 different time allocations for Randoop and JDart that we identified as representative. While our results show no major differences between these different time allocations, we did not fully explore this space and there might be a ratio that would lead to a different outcome. JDart currently cannot symbolically explore native calls, which might lead to not being able to cover program paths (and hence also branches and instructions) that depend on such calls. Our evaluation shows that this indeed happens and that native implementations of methods of the String class in Java are the main culprit, but it does not allow us to provide an estimate of the impact on the achieved code coverage. Finally, while we extensively tested JDoop to make sure it is reliable and performed sanity checks of our results, there is a chance for a bug to have crept in that would influence our results.

Threats to Construct Validity. Here, the main threat is the metrics we used to assess the quality of the generated test suites, and in particular branch coverage in the presence of dead code [3,27]. This threat is reduced by previous work showing that branch coverage performs well as a criterion for comparing test suites [16].

---

[5] http://www.evosuite.org/documentation/measuring-code-coverage.

[6] https://groups.google.com/forum/#!topic/evosuite/ctk2yPIqIoM.

[7] https://stackoverflow.com/questions/41632769/evosuite-code-coverage-does-not-match-with-jacoco-coverage.

6 Related Work

Symbolic Execution. Dynamic symbolic execution [17,36] is a well-known technique implemented by many automatic testing tools (e.g., [6,18,35,43]). For example, SAGE [18] is a white-box fuzzer based on dynamic symbolic execution. It has been routinely applied to large software systems, such as media players and image processors, where it has been successful in finding security bugs. Khurshid et al. [25] extend symbolic execution to support dynamically allocated structures, preconditions, and concurrency.

Several symbolic execution tools specifically target Java bytecode programs. A number of them implement dynamic symbolic execution via Java bytecode instrumentation. JCute [35], the first dynamic symbolic execution engine for Java, uses Soot [39] for instrumentation and lp_solve for constraint solving. CATG [41] uses ASM [2] for instrumentation and CVC4 [10] for constraint solving. Another dynamic symbolic execution engine, LCT [24], supports distributed exploration; it uses Boolector and Yices for solving, but it does not have support for float and double primitive types. A drawback of instrumentation-based tools is that instrumentation at the time of class loading is confined to the SUT. For example, LCT does not by default instrument the standard Java libraries thus limiting symbolic execution only to the SUT classes. Hence, the instrumentation-based tools discussed above provide the possibility of using symbolic models for non-instrumented classes or using pre-instrumented core Java classes.

Several dynamic symbolic execution tools for Java are not based on instrumentation. For example, the dynamic symbolic white-box fuzzer jFuzz [21] is based on Java PathFinder (as is JDart) and can thus explore core Java classes without any prerequisites. Symbolic PathFinder (SPF) [32] is a Java PathFinder extension similar to JDart. In fact, JDart reuses some of the core components of an older version of SPF, notably the solver interface and its implementations. While at its core SPF implements symbolic execution, it can also switch to concrete values in the spirit of dynamic symbolic execution [30]. That enables it to deal with limitations of constraint solvers (e.g., non-linear constraints).

Hybrid Approaches. There are several approaches similar to ours that combine fuzzing or a similar testing technique with dynamic symbolic execution. Garg et al. [15] propose a combination of feedback-directed random testing and dynamic symbolic execution for C and C++ programs. However, they are addressing challenges of a different target language and on a much smaller collection of benchmarks that they simplified before evaluation. The Driller tool [40] interleaves fuzzing and dynamic symbolic execution for bug finding in program binaries, and it targets single-file binaries in search of security bugs. Galeotti et al. [14] apply dynamic symbolic execution in the EvoSuite tool to explore test cases generated with a genetic algorithm. Even though their evaluation is carried out in a different way than the one presented in this paper, the general conclusion is the same in spirit—dynamic symbolic execution does not provide a lot of additional coverage on real-world object-oriented Java software on top of

a random-based test case generation technique. MACE [7] combines automata learning with dynamic symbolic execution to find security vulnerabilities in protocol implementations.

There are other automated hybrid software testing tools that do not strictly combine with symbolic execution (e.g., OCAT [22], Agitator [5], Evacon [19], Seeker [42], DSD-Crasher [8]). Because these tools either focus on a single method at a time or just form random method call sequences, they often fail to drive program execution to hard-to-reach sites in a SUT, which can result in suboptimal code coverage.

Random Testing. Randoop [29] is a feedback-directed random testing algorithm that forms random test cases that are sequences of method calls, while ensuring basic properties such as reflexivity, symmetry, and transitivity. Search-based software testing [28] approaches and tools are gaining traction, which is reflected in four annual search-based software testing tool competitions in recent years [33]. A prominent search-based tool is EvoSuite [13], which combines a genetic algorithm and dynamic symbolic execution. T3 [31] is a tool that generates randomized constructor and method call sequences based on an optimization function. JTExpert [34] keeps track of methods that can change the underlying object and constructs method sequences that are likely to get the object into a desired state. All the search-based testing tools are geared toward testing at the class level, while JDoop performs testing at the application/library level.

Benchmarking Infrastructures. In computer science, any extensive empirical evaluation, software competition, or reproducible research requires a significant software+hardware infrastructure. The Software Verification Competition's BenchExec [4] is a software infrastructure for evaluating verification tools on programs containing properties to verify. It comes with an interface for verification tools to follow, which did not fit our needs: our coverage measurement outcomes cannot be judged in terms of program correctness. The Search-based Software Testing Competition [33] community created an infrastructure for the competition as well. However, just like tools that participate in the competition, their infrastructure is geared toward running a testing tool on just one class at a time. Emulab [45] and Apt [1] are testbeds that provide researchers with an accessible hardware and software infrastructure. They allow for repeatable and reproducible research, especially in the domain of computer systems, by providing an environment to specify the hardware to be used, on top of which users can install and configure a variety of systems.

7 Conclusions

We introduced a hybrid automatic testing approach for object-oriented software, described its implementation JDoop, and performed an extensive empirical exploration of this space. Our approach is an integration of feedback-directed random testing (Randoop) and dynamic symbolic execution (JDart), where random testing performs global exploration, while dynamic symbolic execution performs local exploration (around interesting global states) of a SUT. It is an

iterative algorithm where these two exploration techniques are interleaved in multiple iterations. Our evaluation on real-world object-oriented software shows that dynamic symbolic execution provides consistent improvements in terms of code coverage on top of our baseline (pure feedback-directed random testing) on those examples that are amenable to this method of testing.

References

1. Apt testbed facility. https://www.aptlab.net
2. ASM: A Java bytecode engineering library. http://asm.ow2.org
3. Baluda, M., Denaro, G., Pezzè, M.: Bidirectional symbolic analysis for effective branch testing. IEEE Trans. Softw. Eng. **42**(5), 403–426 (2016)
4. Beyer, D.: Reliable and reproducible competition results with BenchExec and witnesses (report on SV-COMP 2016). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 887–904. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_55
5. Boshernitsan, M., Doong, R., Savoia, A.: From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In: ISSTA, pp. 169–180 (2006)
6. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, pp. 209–224 (2008)
7. Cho, C.Y., Babić, D., Poosankam, P., Chen, K.Z., Wu, E.X., Song, D.: MACE: model-inference-assisted concolic exploration for protocol and vulnerability discovery. In: Proceedings of the 20th USENIX Security Symposium (2011)
8. Csallner, C., Smaragdakis, Y., Xie, T.: DSD-Crasher: a hybrid analysis tool for bug finding. ACM Trans. Softw. Eng. Methodol. **17**(2), 8:1–8:37 (2008)
9. 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). https://doi.org/10.1007/978-3-540-78800-3_24
10. Deters, M., Reynolds, A., King, T., Barrett, C.W., Tinelli, C.: A tour of CVC4: how it works, and how to use it. In: FMCAD, p. 7 (2014)
11. Dimjašević, M., Rakamarić, Z.: JPF-Doop: combining concolic and random testing for Java. In: Java Pathfinder Workshop (JPF) (2013). Extended abstract
12. Eler, M.M., Endo, A.T., Durelli, V.H.S.: Quantifying the characteristics of Java programs that may influence symbolic execution from a test data generation perspective. In: COMPSAC, pp. 181–190 (2014)
13. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: ESEC/FSE, pp. 416–419 (2011)
14. Galeotti, J.P., Fraser, G., Arcuri, A.: Improving search-based test suite generation with dynamic symbolic execution. In: ISSRE, pp. 360–369 (2013)
15. Garg, P., Ivančić, F., Balakrishnan, G., Maeda, N., Gupta, A.: Feedback-directed unit test generation for C/C++ using concolic execution. In: ICSE, pp. 132–141 (2013)
16. Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, M.A., Marinov, D.: Comparing non-adequate test suites using coverage criteria. In: ISSTA, pp. 302–313 (2013)
17. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI, pp. 213–223 (2005)

18. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: whitebox fuzzing for security testing. Queue **10**(1), 20:20–20:27 (2012)
19. Inkumsah, K., Xie, T.: Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In: ASE, pp. 297–306 (2008)
20. JaCoCo Java code coverage library. http://www.jacoco.org/jacoco
21. Jayaraman, K., Harvison, D., Ganesh, V.: jFuzz: a concolic whitebox fuzzer for Java. In: NFM, pp. 121–125 (2009)
22. Jaygarl, H., Kim, S., Xie, T., Chang, C.K.: OCAT: object capture-based automated testing. In: ISSTA, pp. 159–170 (2010)
23. Java PathFinder (JPF). http://babelfish.arc.nasa.gov/trac/jpf
24. Kähkönen, K., Launiainen, T., Saarikivi, O., Kauttio, J., Heljanko, K., Niemelä, I.: LCT: an open source concolic testing tool for Java programs. In: BYTECODE, pp. 75–80 (2011)
25. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_40
26. Luckow, K., Dimjašević, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamarić, Z., Raman, V.: JDart: a dynamic symbolic analysis framework. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 442–459. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_26
27. Marcozzi, M., Bardin, S., Kosmatov, N., Papadakis, M., Prevosto, V., Correnson, L.: Time to clean your test objectives. In: ICSE, pp. 456–467 (2018)
28. McMinn, P.: Search-based software testing: past, present and future. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, pp. 153–163 (2011)
29. Pacheco, C., Lahiri, S., Ernst, M., Ball, T.: Feedback-directed random test generation. In: ICSE, pp. 75–84 (2007)
30. Pasareanu, C.S., Rungta, N., Visser, W.: Symbolic execution with mixed concrete-symbolic solving. In: ISSTA, pp. 34–44 (2011)
31. Prasetya, I.S.W.B.: Budget-aware random testing with T3: benchmarking at the SBST2016 testing tool contest. In: SBST, pp. 29–32 (2016)
32. Păsăreanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: ISSTA, pp. 15–26 (2008)
33. Rueda, U., Just, R., Galeotti, J.P., Vos, T.E.J.: Unit testing tool competition – round four. In: SBST, pp. 19–28 (2016)
34. Sakti, A., Pesant, G., Guéhéneuc, Y.G.: JTExpert at the fourth unit testing tool competition. In: SBST, pp. 37–40 (2016)
35. Sen, K., Agha, G.: CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_38
36. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/FSE, pp. 263–272 (2005)
37. The SF110 benchmark suite, July 2013. http://www.evosuite.org/experimental-data/sf110
38. Shafiei, N., van Breugel, F.: Automatic handling of native methods in Java PathFinder. In: SPIN, pp. 97–100 (2014)
39. Soot: A Java optimization framework. http://sable.github.io/soot

40. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: augmenting fuzzing through selective symbolic execution. In: NDSS (2016)
41. Tanno, H., Zhang, X., Hoshino, T., Sen, K.: TesMa and CATG: automated test generation tools for models of enterprise applications. In: ICSE, pp. 717–720 (2015)
42. Thummalapenta, S., Xie, T., Tillmann, N., de Halleux, J., Su, Z.: Synthesizing method sequences for high-coverage testing. SIGPLAN Not. **46**(10), 189–206 (2011)
43. Tillmann, N., de Halleux, J.: Pex–white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79124-9_10
44. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003)
45. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. SIGOPS Oper. Syst. Rev. **36**(SI), 255–270 (2002)

Making Linearizability Compositional for Partially Ordered Executions

Simon Doherty[1], Brijesh Dongol[2(✉)], Heike Wehrheim[3], and John Derrick[1]

[1] University of Sheffield, Sheffield, UK
[2] University of Surrey, Guildford, UK
b.dongol@surrey.ac.uk
[3] University of Paderborn, Paderborn, Germany

Abstract. In the interleaving model of concurrency, where events are totally ordered, linearizability is compositional: the composition of two linearizable objects is guaranteed to be linearizable. However, linearizability is not compositional when events are only partially ordered, as in the weak-memory models that describe multicore memory systems. In this paper, we present a generalisation of linearizability for concurrent objects implemented in weak-memory models. We abstract from the details of specific memory models by defining our condition using Lamport's execution structures. We apply our condition to the C11 memory model, providing a correctness condition for C11 objects. We develop a proof method for verifying objects implemented in C11 and related models. Our method is an adaptation of simulation-based methods, but in contrast to other such methods, it does not require that the implementation totally orders its events. We apply our proof technique and show correctness of the Treiber stack that blocks on empty, annotated with C11 release-acquire synchronisation.

1 Introduction

Linearizability [23, 24] is a well-studied [16] condition that defines correctness of a concurrent object in terms of a sequential specification. It ensures that for each history (i.e., execution trace) of an implementation, there is a history of the specification such that (1) each thread makes the same method invocations in the same order, and (2) the order of non-overlapping operation calls is preserved. The condition, however, critically depends on the existence of a total order of memory events (e.g., as guaranteed by *sequential consistency* (SC) [31]) to guarantee contextual refinement [20] and compositionality [24]. Unfortunately, most modern execution environments can only guarantee a partial order of memory events, e.g., due to the effects of relaxed memory [3, 5, 8, 34]. It is known that a naive adaptation of linearizability to the partially ordered setting of weak memory is problematic from the perspective of contextual refinement [18]. In this paper, we propose a generalisation of linearizability to cope with partially ordered executions, which we show satisfies compositionality.

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 110–129, 2018.
https://doi.org/10.1007/978-3-319-98938-9_7

Init: $x = 0, y = 0$

Process 1	Process 2
$1 : x := 1;$	$1 :$ `if`$(y = 1)$
$2 : y := 1;$	$2 :$ `assert`$(x = 1);$

Fig. 1. Writing to shared variables

Init: $\mathrm{S} = \langle\rangle, \mathrm{S}' = \langle\rangle$

Process 1	Process 2
1 : S.Push(1);	1 : `if`(S'.Pop = 1)
2 : S'.Push(1);	2 : `assert`(S.Pop = 1);

Fig. 2. Writing to shared stacks

To motivate the problem consider the following. Figures 1 and 2 show two examples[1] of multi-threaded programs on which weak memory model effects can be observed. Figure 1 shows two threads writing to and reading from two shared variables x and y. Under SC, the `assert` in process 2 never fails: if y equals 1, x must also equal 1. However, this is not true in weak memory models like C11 [8,28]: if the writes to x and y are *relaxed*, process 2 may observe the write to y, yet also observe the initial value x (missing the write to x by process 1).

Such effects are not surprising to programmers familiar with memory models [8,28]. However, programmer expectations for linearizable objects, even in a weak memory model like C11, are different: if the two stacks S and S′ in Fig. 2 are linearizable, the expectation is that the `assert` will never fail since linearizable objects are expected to be *compositional* [23,24], i.e., any combination of linearizable objects must itself be linearizable. However, it is indeed possible for the two stacks to be linearizable (using the classical definition), yet for the program to generate an execution in which the `assert` fails, i.e., the composition of the two stacks is not linearizable. The issue here is that linearizability, when naively applied to a weak memory setting, allows too many operations to be considered "unordered".

Failure of compositionality is repaired by strengthening the requirements of linearizability on partial orders. Namely, we require the ability to infer enough order in an execution to ensure that the method call S.Push(1) *precedes* S.Pop, forcing S.Pop to return 1, whenever S'.Push(1) precedes S'.Pop.

The contributions of this paper are as follows.

- Our main contribution is the development of a new compositional notion of correctness; we call this condition *causal linearizability*.
- We establish two meta-theoretical properties of causal linearizability. First, we show that, as expected, causal linearizability reduces to linearizability when the underlying memory model is totally ordered. Second, we show that that causal linearizability is the weakest possible strengthening of linearizability that guarantees compositionality, i.e., any correctness condition stronger than linearizability that is also compositional must imply causal linearizability.
- We present a new inductive simulation-style proof technique for verifying causal linearizability of weak memory implementations of concurrent objects, where the induction is over linear extensions of the happens-before relation. This is the first such proof method for weak memory, and one of the first

[1] Example in Fig. 2 inspired by H.-J. Boehm talk at Dagstuhl, Nov. 2017.

that enables refinement-based verification, building on existing techniques for linearizability in SC [13,16,36].
– We apply this proof technique and verify causal linearizability of a blocking version of the Treiber Stack executing in the C11 weak memory model. For the standard Treiber Stack under C11, we identify a synchronisation pitfall when using only release-acquire synchronisation.

Causal linearizability is so called because it takes into account the causal relationship between events in a way that is relevant to weak memory models. There is an earlier definition of a condition also called “causal linearizability” introduced by Doherty and Derrick in [11]. However, this earlier definition considers causality at the level of (interleaved) sequences and only applies to memory models such as TSO, that satisfy certain operational properties.[2] In contrast, the definition in this paper (Definition 6) considers causality directly over partial orders, making it applicable to a wider range of memory models.

The definition of causal linearizability in this paper is built on the same concerns as the earlier definition in [11], but is not a generalisation of it in a technical sense. Thus Definition 6 does not reduce to the condition in [11], or vice versa, although both reduce to classical linearizability [24]. All mentions of “causal linearizability” in this paper refers to Definition 6. Further comparisons to related correctness conditions are given in Sect. 8.

Causal linearizability is defined in terms of an *execution structure* [32], taking two different relations over operations into account: a “precedence order” (describing operations that are ordered in real time) and a “communication relation”. Execution structures allow one to infer the additional precedence orders from communication orders (see Definition 3 (A5)). Applied to Fig. 2, for a weak memory execution in which the `assert` fails, the execution restricted to stack S would not be causally linearizable in the first place (see Sect. 3 for full details). Execution structures are generic, and can be constructed for any weak memory execution that includes method invocation/response events. We develop one such scheme for mapping executions to execution structures based on the *happens-before* relation of the C11 memory model.

This paper is structured as follows. We present our motivating example, the Treiber Stack in C11 in Sect. 2; describe the problem of compositionality and motivate our execution-structure based solution in Sect. 3; and formalise causal linearizability and compositionality in Sect. 4. Causal linearizability for C11 is presented in Sect. 5, and verification of the stack described in Sect. 6. Section 7 describes a synchronisation pitfall.

2 Treiber Stack in C11

The example we consider (see Algorithm 1) is the Treiber Stack [40] (well-studied in a SC setting, but not in a weak memory one), executing in a recent version of the C11 [30] memory model. In C11, commands may be annotated, e.g., R

[2] In retrospect, the name “causal linearizability” is more fitting to this current paper.

Algorithm 1. Release-Acquire Treiber Stack

```
1: procedure Init
2:     Top := null;

3: procedure Push(v)
4:     n := new(node) ;
5:     n.val := v ;
6:     repeat
7:         top :=^A Top ;
8:         n.nxt := top ;
9:     until CAS^R(&Top, top, n)

10: function Pop
11:     repeat
12:         repeat
13:             top :=^A Top ;
14:         until top ≠ null ;
15:         ntop := top.nxt ;
16:     until CAS^R(&Top, top, ntop)
17:     return top.val ;
```

(for release) and A (for acquire), which introduces extra synchronisation, i.e., additional order over memory events [8,28]. We assume racy read and write accesses that are not part of an annotated command are *unordered* or *relaxed*, i.e., we do not consider the effects of non-atomic operations [8]. Full details of the C11 memory model are deferred until Sect. 5.

Due to weak memory effects, the events under consideration, including method invocation and response events are partially ordered [5,6,14,28,30]. As we show in Sect. 3, it turns out that one cannot simply reapply the standard notion of linearizability in this weaker setting; compositionality demands that we use modified correctness condition, causal linearizability, that additionally requires "communication" across conflicting operations.

In Algorithm 1, all accesses to Top are via an annotated command. Thus, any read of Top (lines 7, 13) reading from a write to Top (lines 9, 16) induces *happens-before order* from the write to the read. This order, it turns out, is enough to guarantee invariants that are in turn strong enough to guarantee[3] causal linearizability of the Stack (see Sect. 6).

Note that we modify the Treiber Stack so that the Pop operation blocks by spinning instead of returning empty. This is for good reason - it turns out that the standard Treiber Stack (with a non-blocking Pop operation) is *not* naturally compositional if the only available synchronisation is via release-acquire atomics (see Sect. 7).

3 Compositionality and Execution Structures

This section describes the problems with compositionality for linearizability of concurrent objects under weak execution environments (e.g., relaxed memory) and motivates a generic solution using *execution structures* [32].

[3] Note that a successful CAS operation comprises both a read and a write access to Top, but we only require release synchronisation here. The corresponding acquire synchronisation is provided via the earlier read in the same operation. This synchronisation is propagated to the CAS by *sequenced-before* (aka program order), which, in C11, is included in happens-before (see Sect. 6 for details).

Notation. First we give some basic notation. Given a set X and a relation $r \subseteq X \times X$, we say r is a *partial order* iff it is reflexive, antisymmetric and transitive, and a *strict order*, iff it is irreflexive, antisymmetric and transitive. A partial or strict order r is a *total order on* Xiff either $(a, b) \in r$ or $(b, a) \in r$ for all $a, b \in X$. We typically use notation such as $<$, $\prec$, $\longrightarrow$ to denote orders, and write, for example, $a < b$ instead of $(a, b) \in <$. We let X^* denote the set of all finite sequences over X, let $\langle\rangle$ denote the empty sequence and use $\circ$ as a concatenation operator on sequences. For a sequence w, we let $\twoheadrightarrow_w$ be the (total) order on its elements: $e \twoheadrightarrow_w e'$ if $w = w_1 \circ \langle e \rangle \circ w_2 \circ \langle e' \rangle \circ w_3$.

Fix a set of *invocations* Inv and a set of *responses* Res. A pair from $Inv \times Res$ represents an *operation*. Each invocation includes both a *method name* and any arguments passed to the method; each response includes any values returned from the method. For example, for a stack S of natural numbers, the *invocations* of the stack might be represented by the set $\{\textsc{Push}(n) \mid n \in \mathbb{N}\} \cup \{\textsc{Pop}\}$, and the *responses* by $\mathbb{N} \cup \{\bot, \texttt{empty}\}$, and the set of operations of the stack is

$$\Sigma_{\mathrm{S}} = \{(\textsc{Push}(n), \bot), (\textsc{Pop}, n) \mid n \in \mathbb{N}\} \cup \{(\textsc{Pop}, \texttt{empty})\}.$$

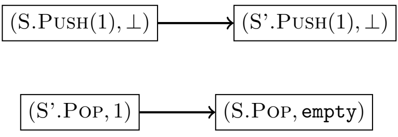
In an execution, an occurrence of an invocation, response, or operation will take the form of an *event*. In a full treatment, events would have the form $e = (l, t, g)$, where l is a *label* of type $Inv \cup Res$ (for executions of a concrete implementation) or $Inv \times Res$ (for executions of an abstract specification), t is a *thread identifier* t from some given set of threads (or processes) Tid and g is a *tag* uniquely identifying the event in the execution. However, for clarity of presentation, we omit tags in this paper. Furthermore, for uniformity, we assume that all invocations, responses and operations are indexed by thread identifiers. For example, the invocations are now given by the set $\{\textsc{Push}_t(n), \textsc{Pop}_t \mid t \in T \wedge n \in \mathbb{N}\}$. We only make thread ids explicit when necessary. We let $tid(e)$ be the thread identifier of event e. For a sequence or partial order of events w, we let $w{\restriction}_t$ be the restriction of w to events e with $tid(e) = t$ only.

The standard notion of linearizability is defined for a concurrent history, which is a sequence (or total order) of *invocation* and *response* events of operations. Since operations are concurrent, an invocation of an operation may not be directly followed by its matching response in this sequence, and hence, a history induces a partial order on operations (through the total order on events). For linearizability, we focus on the *precedence* order (denoted $\longrightarrow$), where, for operations o and o', we say $o \longrightarrow o'$ in a history iff the response of operation o occurs before the invocation of operation o' in the history. A concurrent implementation of an object is linearizable if the precedence order ($\longrightarrow$) for *any* history of the object can be extended to a total order that is *legal* for the object's specification [24]. It turns out that linearizability in this setting is *compositional* [23,24]: any history of a family of linearizable objects is itself guaranteed to be linearizable.

Unfortunately, histories in modern execution contexts (e.g., due to relaxed memory or distributed computation) are only partially ordered since processes do not share a single global view of time. It might seem that this is unproblematic for linearizability and that the standard definition can be straightforwardly applied

to this weaker setting. However, it turns out that a naive application fails to satisfy *compositionality*. To see this, consider the following example.

Example 1. Consider an execution of Fig. 2 where the operations are only ordered by a *happens-before* relation, which is a relation present in many weak-memory models [3,5,8,34]. Since we do not have a global notion of time, we say operation o precedes o' (denoted $o \longrightarrow o'$) if the response of o happens before the invocation of o' (also see [18]). For the C11 memory model, happens-before includes program order, and hence, the progam in Fig. 2 may generate the following execution, where operations executed by the same thread are precedence ordered.

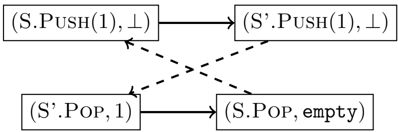


If we restrict the execution above to S only, we can obtain a legal stack behaviour by linearizing (S.Pop, empty) before (S.Push(1), $\perp$) without contradicting the precedence order $\longrightarrow$ in the diagram above. Similarly, the execution when restricted to S$'$ is linearizable. However, the full execution is not linearizable: ordering the pop of S before its push, and the push of S' before its pop contradicts the precedence order $\longrightarrow$. $\square$

A key contribution of this paper is the development of a correctness condition, *causal linearizability*, that recovers compositionality of concurrent objects with partially ordered histories. Our definition is based on two main insights.

Our first insight is that one must augment the precedence order with additional information about the underlying concurrent execution. In particular, one must introduce information about the *communication* between operations, e.g., when one operation sees the effects of another one. In our example, a pop would see the effect of a push; in the Treiber algorithm it would specifically see the change of Top. Causal linearizability states that the ordering we impose during linearization has to (a) preserve the precedence order of operations and (b) has to be consistent with the communication order. We represent communication by a relation $\dashrightarrow$.

Example 2. Consider again the partial order in Example 1. For stack S, we must linearize pop before push, and for stack S', push before pop. Causal linearizability mandates the existence of a *logical order* that contains $\longrightarrow$ such that all linear extensions of the logical order are legal w.r.t. the specification object. Moreover, it requires that this logical order is contained within the communication relation. Hence, in Example 1, neither S nor S' is causally linearizable: for S, the only valid logical order is S.Pop before S.Push(1), but there is currently no communication from S.Pop to S.Push(1). Thus, the execution in Example 1 is not a counterexample to compositionality of causal linearizability. Now consider changing the example by introducing communication as follows:



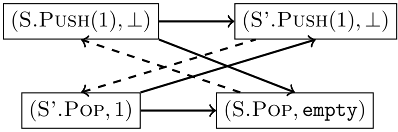
Here, communication is introduced in a manner consistent with the logical order, which requires that (S.Pop, `empty`) is linearized before (S.Push(1), $\perp$) and that (S'.Push(1), $\perp$) is linearized before (S'.Pop, 1). So far, we would consider this to be a valid counterexample to compositionality. We describe why this cannot be the case below. □

Our second insight is that the operations (taken as events) together with the precedence order $\longrightarrow$ and the communication relation $\dashrightarrow$ must form an *execution structure* [32].

Definition 3 (Execution structure). *If E is a finite*[4] *set of events, and $\longrightarrow, \dashrightarrow \subseteq E \times E$ are relations over E (the* precedence order *and* communication relation*), an* execution structure *is a tuple $(E, \longrightarrow, \dashrightarrow)$ satisfying the following axioms for $e_1, e_2, e_3 \in E$.*

A1. *The relation $\longrightarrow$ is a strict order.*
A2. *Whenever $e_1 \longrightarrow e_2$, then $e_1 \dashrightarrow e_2$ and $\neg(e_2 \dashrightarrow e_1)$.*
A3. *If $e_1 \longrightarrow e_2 \dashrightarrow e_3$ or $e_1 \dashrightarrow e_2 \longrightarrow e_3$, then $e_1 \dashrightarrow e_3$.*
A4. *If $e_1 \longrightarrow e_2 \dashrightarrow e_3 \longrightarrow e_4$, then $e_1 \longrightarrow e_4$.* □

Example 4. We apply Definition 3 to Example 2. The requirements of an execution structure, in particular axiom **A4** necessitate that we introduce additional precedence order edges $\longrightarrow$ as follows.



For example, the edge (S'.Pop, 1)$\longrightarrow$(S'.Push(1), $\perp$) is induced by the combination of edges (S.Push(1), $\perp$)$\longrightarrow$(S'.Push(1), $\perp$)$\dashrightarrow$(S'.Pop, 1)$\longrightarrow$(S.Pop, `empty`) together with axiom **A4**. However, this structure now fails to satisfy axiom **A2** and is thus no longer a proper execution structure. □

Hence, for our running example, compositionality no longer fails. We conclude that for causally linearizable stacks, the `assert` in Fig. 2 always holds if it is executed.

[4] The original presentation allows infinite execution structures but requires that $\longrightarrow$ be well founded.

4 Causal Linearizability

Causal linearizability extends linearizability to cope with partially ordered executions. Next, we will formally define it and its compositionality property.

Like ordinary linearizability, causal linearizability is defined by comparing the behaviour of concurrent executions to *legal* sequential ones. The comparison typically proceeds by bringing concurrent operations in sequence under some given constraints. This basic principle is kept for the partially ordered case. Legality is defined by a *sequential object* specification, which we define operationally.

Definition 5. *A* sequential object *is a 4-tuple* $(\Sigma, \Gamma, init, \tau)$*, where* Σ *is a set of* labels*,* Γ *is a set of* states*,* $init \in \Gamma$ *is an* initial state*, and* $\tau \subseteq \Gamma \times \Sigma \times \Gamma$ *is a* transition relation*.*

The set $\Sigma \subseteq Inv \times Res$ consists of pairs of invocations and responses. For our stack example, $\Gamma = \mathbb{N}^*$, $init = \langle\rangle$ and

$$\tau = \{(s, (\text{PUSH}(n), \bot), \langle n \rangle \circ s) \mid n \in \mathbb{N}\} \cup \{(\langle n \rangle \circ s, (\text{POP}, n), s) \mid n \in \mathbb{N}\} \\ \cup \{(\langle\rangle, (\text{POP}, \texttt{empty}), \langle\rangle)\}$$

We write $s \overset{op}{\rightsquigarrow}_\tau s'$ for $(s, op, s') \in \tau$. For a sequence $w \in \Sigma^*$, we write $s \overset{w}{\rightsquigarrow}_\tau s'$ iff either $w = \langle\rangle$ and $s = s'$, or $w = \langle op \rangle \circ w'$ and there exists an s'' such that $s \overset{op}{\rightsquigarrow}_\tau s''$ and $s'' \overset{w'}{\rightsquigarrow}_\tau s'$. The set of *legal histories* of an object $\mathbb{S} = (\Sigma, \Gamma, init, \tau)$ is given by $legal_{\mathbb{S}} = \{w \in \Sigma^* \mid \exists s \in \Gamma.\ init \overset{w}{\rightsquigarrow}_\tau s\}$.

In general, executions of concurrent processes might invoke operations on more than one object. To capture this, we define a notion of an *object product.* If $\mathbb{S}_1 = (\Sigma_1, \Gamma_1, init_1, \tau_1)$ and $\mathbb{S}_2 = (\Sigma_2, \Gamma_2, init_2, \tau_2)$ are two sequential objects with $\Sigma_1 \cap \Sigma_2 = \emptyset$, the object product of $\mathbb{S}_1$ and $\mathbb{S}_2$ is defined by $\mathbb{S}_1 \otimes \mathbb{S}_2 = (\Sigma_1 \cup \Sigma_2, \Gamma_1 \times \Gamma_2, (init_1, init_2), \tau_1 \otimes \tau_2)$, where

$$\tau_1 \otimes \tau_2 = \{((s_1, s_2), op_1, (s_1', s_2)) \mid op_1 \in \Sigma_1 \wedge (s_1, op_1, s_1') \in \tau_1\} \\ \cup \{((s_1, s_2), op_2, (s_1, s_2')) \mid op_2 \in \Sigma_2 \wedge (s_2, op_2, s_2') \in \tau_2\}.$$

Clearly, this construction can be generalised to products of more than two objects, provided their sets of actions are pairwise disjoint. We abstain from such a treatment here since a compositionality result for two objects is sufficient to ensure compositionality over multiple objects.

Causal linearizabilty compares the concurrent execution given by an execution structure to the legal sequential behaviour. The constraint on this sequentialization is that the precedence and the communication order of execution structures provide lower and upper bounds for the allowed ordering. More precisely, we use a partial order $<$ that contains all orders necessary to ensure legality, i.e., there is no linear extension of $<$ that is not legal. Causal linearizability requires (a) the precedence order of execution structures to be preserved by this order, and (b) this order to be contained in the communication relation. We say a strict partial order $<$ is a *logical order* of an execution structure

$\mathbb{E} = (E, \longrightarrow, \dashrightarrow)$ iff $< \subseteq E \times E$ and $\longrightarrow \subseteq < \subseteq \dashrightarrow$. For concurrent objects, one possible instantiation of a logical order is given in [12], where the logical order corresponds to a conflict-based notion of causality.

For a partial order $< \subseteq E \times E$, we let $LE(<) = \{w \in E^* \mid < \subseteq \twoheadrightarrow_w\}$ be the set of *linear extensions* of $<$.

Definition 6. *Let $\mathbb{S}$ be a sequential object. An execution structure $\mathbb{E}$ is* causally linearizable *w.r.t. $\mathbb{S}$ iff there exists a logical order $<$ of $\mathbb{E}$ such that $LE(<) \subseteq legal_{\mathbb{S}}$.*

Causal linearizability guarantees compositionality, i.e., the composition of causally linearizable concurrent objects is causally linearizable. More formally, for an execution structure $\mathbb{E} = (E, \longrightarrow, \dashrightarrow)$ and events $X \subseteq E$, we let $\mathbb{E}\downharpoonright_X$ be the execution structure restricted to X, i.e., $(X, \longrightarrow \cap (X \times X), \dashrightarrow \cap (X \times X))$.

Theorem 7 (Compositionality). *If $\mathbb{S}_1 = (\Sigma_1, \dots)$ and $\mathbb{S}_2 = (\Sigma_2, \dots)$ are sequential objects with $\Sigma_1 \cap \Sigma_2 = \emptyset$ and $\mathbb{E} = (E, \longrightarrow, \dashrightarrow)$ is an execution structure, then $\mathbb{E}\downharpoonright_{\Sigma_1}$ is causally linearizable w.r.t. $\mathbb{S}_1$ and $\mathbb{E}\downharpoonright_{\Sigma_2}$ causally linearizable w.r.t. $\mathbb{S}_2$ iff $\mathbb{E}$ is causally linearizable w.r.t. $\mathbb{S}_1 \otimes \mathbb{S}_2$.* □

Standard linearizability as introduced by Herlihy and Wing [24] is defined on executions (histories) which are totally ordered sequences of invocations and responses of operations, i.e. a history h is an element of $(Inv \cup Res)^*$. Note that this allows executions in which operations are concurrent because invocation and response events are now separated. Histories are required to be *well-formed* and *complete*[5], which means that the projection of a history onto one thread forms a sequence of invocations and corresponding responses.

A strict order $\prec$ on $Inv \cup Res$ is well-formed and complete iff for all threads $t \in Tid$, $\prec\downharpoonright_t$ is *sequential*, i.e., forms an alternating total order of invocations and responses starting with an invocation. Invocations and responses thus form *matching pairs* as defined by a function mp. For a strict order $\prec$ such that $i \prec r$, $(i, r) \in mp(\prec)$ iff $tid(i) = tid(r)$ and there is no event e such that $i \prec e \prec r$ and $tid(e) = tid(i)$. This allows us to derive an execution structure from any strict order and thus also from a history h by using its ordering $\twoheadrightarrow_h$.

Definition 8. *Let $\prec$ be a well-formed and complete strict order on $Inv \cup Res$. We say $exec(\prec) = (E, \longrightarrow, \dashrightarrow)$ is the execution structure corresponding to $\prec$ if*

$$\begin{array}{rcl} E & = & mp(\prec), \\ \longrightarrow & = & \{((i_1, r_1), (i_2, r_2)) \in E \times E \mid r_1 \prec i_2\}, \\ \dashrightarrow & = & \{((i_1, r_1), (i_2, r_2)) \in E \times E \mid i_1 \prec r_2\}. \end{array}$$

Note that this construction guarantees a *saturation* property: for two events e, e', we either have $e \longrightarrow e'$ or $e' \dashrightarrow e$.

The classical definition of linearizability only employs the precedence ordering of an execution structure. That is, $(E, \longrightarrow, \dashrightarrow)$ is linearizable w.r.t. a sequential

[5] Note that we only assume completeness for the sake of simplicity here.

object $\mathbb{S}$ iff there exists a sequence $hs \in legal_{\mathbb{S}}$ such that (i) $\longrightarrow\!\restriction_t = \twoheadrightarrow_{hs}\restriction_t$ for all $t \in Tid$ (threads execute the same sequence of operations) and (ii) $\longrightarrow \subseteq \twoheadrightarrow_{hs}$ (precedence ordering between operations is preserved). We say that a strict order $\prec$ is linearizable iff $exec(\prec)$ is linearizable and that a history h is linearizable iff $\twoheadrightarrow_h$ is linearizable.

Theorem 9. *Suppose h is a history and $\mathbb{S}$ a sequential object. Then h is linearizable w.r.t. $\mathbb{S}$ iff $exec(\twoheadrightarrow_h)$ is causally linearizable w.r.t. $\mathbb{S}$.* □

We now provide an adequacy result for causal linearizability, i.e., show that causal linearizability is, in fact, the *weakest* possible strengthening of linearizability that is compositional. The technical exposition is formalised in terms of correctness conditions that guarantee linearizability. Here, we regard a correctness condition to be a function mapping a sequential object to the set of all well-formed strict orders on $Inv \cup Res$ accepted as being correct for the object, where the mapping is closed under renaming of the operations.

We let $\mathcal{S}$ be the set of all possible sequential objects and $\mathcal{H}$ the set of all possible well-formed complete strict orders on $Inv \cup Res$. To formalise closure under renaming, we assume a bijection $b : X \to Y$ between sets X and Y. If $\mathbb{S} = (X, \Gamma, init, \tau)$ is a sequential object, define $b[\mathbb{S}] = (Y, \Gamma, init, b[\tau])$, where $b[\tau] = \{(s, b(x), s') \mid (s, x, s') \in \tau\}$ and if $w \in X^*$, define $b[w] \in Y^*$ to be the sequence obtained from w by replacing each w_i by $b(w_i)$.

Definition 10. *We say a function $\Delta : \mathcal{S} \to 2^{\mathcal{H}}$ is a* correctness condition *iff Δ is closed under renaming of operations, i.e., for all bijective functions $b : \Sigma \to \Sigma'$ and for all $\mathbb{S} = (\Sigma, \dots) \in \mathcal{S}$, we have $\prec \in \Delta(\mathbb{S})$ iff $b[\prec] \in \Delta(b[\mathbb{S}])$.*

Definition 11. *We say $\Delta : \mathcal{S} \to 2^{\mathcal{H}}$* guarantees linearizability *iff for all $\mathbb{S} \in \mathcal{S}$, each $\prec \in \Delta(\mathbb{S})$ is linearizable w.r.t. $\mathbb{S}$.*

Our adequacy result for causal linearizability is defined for well-formed strict orders that have exactly one possible legal linearization. Formally, for a correctness condition Δ and sequential object $\mathbb{S}$, we say $\prec \in \Delta(\mathbb{S})$ is *strongly synchronised* iff it is linearizable w.r.t. exactly one $hs \in legal_{\mathbb{S}}$.

Theorem 12. *Let Δ be a compositional correctness condition, and let $\mathbb{S}$ be a sequential object. Then for any strongly synchronised strict order $\prec \in \Delta(\mathbb{S})$, $exec(\prec)$ is causally linearizable.*

Strong synchronisation may always arise for some specifications, e.g., a data structure such as a stack or a counter object that only provides a *fetch-and-increment* operation. In general, the execution of any object may be strongly synchronised due to interactions with other objects or a client (see [18]), causing additional precedence order to be introduced. For example, a client thread of a concurrent object may introduce precedence order via program order, inserting fences between operation calls, or calling objects that induce additional order [6, 18]. Thus for typical sequential objects, a correctness condition that prohibited strongly synchronized executions would be overly restrictive. Theorem 12 ensures that, for such executions, if the correctness condition is compositional then it is at least as strong as causal linearizability.

5 C11 Executions

We now briefly introduce the C11 memory model as to be able to reason about programs executing within C11. To this end, we simply give a (condensed) adaptation of the programming-language oriented presentation of C11 [14,28], but we ignore various features of C11 not needed for our discussion, including non-atomic operations and fences. For a more complete explanation see e.g. [28].

The C11 Memory-Model. The memory model specifies how read and write operations access shared state. Let L be a set of such shared *locations* (ranged over by x, y) and let V be a set of values (ranged over by u, v). Our model employs a set of *memory events*, which can be partitioned into *read* events, R, *write* events, W, and *update* (read-modify-write) events, U. A read event would e.g. take the form $rd(x, 0)$. An update event occurs for instance when a CAS operation is executed: a shared location is read, compared to a local variable and then possibly written to. We let $Mod = W \cup U$ be the set of events that *modify* a location, and $Qry = R \cup U$ be the set of events that *query* a location. For any memory event e, let $loc(e)$ be the event's accessed location and $Loc(x) = \{e \mid loc(e) = x\}$ the set of events accessing location x. For any query event let $rval(e)$ be the value read; and for any modification event let $wval(e)$ be the value written. An event may carry a *synchronisation annotation*, which (in our restricted setting) may either be a release, R, or an acquire, A, annotation, and we let $ann(e)$ be an event e's annotation.

Definition 13. *A* C11 execution *is a tuple* $\mathbb{D} = (D, sb, rf, mo)$*, where* D *is a set of events, and* $sb, rf, mo \subseteq D \times D$ *define the* sequenced-before*,* reads-from *and* modification order *relations, respectively.*

We say a C11 execution is *valid* when it satisfies the following constraints: **(V1)** sb is a strict order, such that, for each process p, the projection of sb onto p is a total order; the reads-from relation specifies the write a particular read event reads-from: $rf \subseteq Mod \times Qry$ and **(V2)** for all $(w, r) \in rf$, $loc(w) = loc(r)$ and $wval(w) = rval(r)$ as well as **(V3)** for all $r \in D \cap Qry$, there exists some $w \in D \cap Mod$ such that $(w, r) \in rf$; the modification order relates writes to the same location and these writes are totally ordered: **(V4)** for all $(w, w') \in mo$, $loc(w) = loc(w')$; and **(V5)** for all $w, w' \in Mod$ such that $loc(w) = loc(w')$, $(w, w') \in mo$ or $(w', w) \in mo$.

Other relations can be derived from these basic relations. For example, assuming D_{R} and D_{A} denote the sets of events with release and acquire annotations, respectively, the *synchronises-with* relation, $sw = rf \cap (D_{\mathsf{R}} \times D_{\mathsf{A}})$, creates interthread ordering guarantees based on synchronisation annotations. The annotations R and A can thus be used by programmers to achieve certain visibility effects of their write events. The *from-read* relation, $fr = (rf^{-1}; mo) \backslash Id$, relates each query to the events in modification order *after* the modification that it read from. Our final derived relation is the *happens before* relation $hb = (sb \cup sw)^+$, which formalises causality. We say that a C11 execution is *consistent* if **(C1)** hb is acyclic, and **(C2)** $hb; (mo \cup rf \cup fr)$ is irreflexive.

Method Invocations and Responses. So far, the events appearing in our memory model are low level read and write events. Our goal is to model algorithms such as the Treiber stack. Thus, we add *method events* to the standard model, namely, *invocations*, Inv, and *responses*, Res. Unlike weak memory at the processor architecture level, where program order may not be preserved [18], program order in C11 is consistent with happens-before order, and hence, invocation and response events can be introduced here in a straightforward manner. The only additional requirement is that validity also requires **(V6)** sb for each process projected restricted $Inv \cup Res$ must be an alternating sequence of invocations and matching responses, starting with an invocation. In any execution of a well-formed program, this condition is naturally satisfied.

From C11 Executions to Execution Structures. A C11 execution with method invocations and responses naturally gives rise to an execution structure. First, for a C11 execution $\mathbb{D}$ and $IR = Inv \cup Res$, we let $hb_{ir} = hb \cap (IR \times IR)$, i.e., the happens-before relation of $\mathbb{D}$ restricted to the invocation and response events. By **(V6)**, hb_{ir} is well-formed and complete. Thus, we can apply the construction defined in Sect. 4 to build an execution structure $exec(hb_{ir})$.

Definition 14. *We say that a C11 execution* $\mathbb{D}$ *is* causally linearizable *w.r.t. a sequential object if* $exec(hb_{ir})$ *is.* □

Compositionality of causal linearizability thus naturally carries over to C11 executions. Finally, we say that a data structure (like the Treiber stack) is causally linearizable on C11 when all its C11 executions are. Thus, we will in the following investigate how we can prove such a property.

6 Verification

We now describe an operational method for proving that a given C11 execution is causally linearizable w.r.t. a given sequential object. Our method is based on a simulation-based proof rule described in Sect. 6.1. We illustrate our technique on the Treiber Stack (Sect. 6.2), which is often used as an exercise in the verification literature [16]. Unlike these existing verifications, we consider weak memory, and hence, the stack in Algorithm 1 generates more behaviours than in a standard sequentially consistent setting. The proof in Sect. 6.2 below is the first to verify that the stack under C11 satisfies causal linearizability. Moreover, our proof technique, which considers simulation over a happens-before relation, is novel to this paper.

6.1 A Simulation Relation over Happens-Before

For the remainder of this section, fix a C11 execution $\mathbb{D} = (D, sb, rf, mo)$, and a sequential object $\mathbb{S} = (\Sigma, \Gamma, init, \tau)$. We describe a method for proving that

$\mathbb{D}$ is causally linearizable w.r.t. $\mathbb{S}$. In what follows, we write $e \longrightarrow_{hb} e'$ when $(e, e') \in hb_{\mathbb{D}}$.

As in the interleaving setting our method depends on assigining *linearization points* [16] to each operation. Therefore, the verifier must define a function $lp : D \cap Inv \to D$, which returns the memory event that linearizes the given high-level operation, represented by its invocation. For simplicity, in this presentation, we require that our linearizations be injective.[6] Recall from the previous section that the operations in the execution structure $exec(hb_{ir})$ are elements of matching pairs from the set $mp(hb_{ir})$. To recover the abstract order of operations corresponding to a linearization, we use $<_{lp} = \{((i,r),(i',r')) \mid lp(i) \longrightarrow_{hb} lp(i') \wedge (i,r),(i',r') \in mp(hb_{ir})\}$.

Definition 15. *We say lp is a* linearization *iff for each* $i \in D \cap Inv$, *and matching response* r, $i \longrightarrow_{hb} lp(a) \longrightarrow_{hb} r$. *Furthermore, we say lp is a* legal linearization *iff* $LE(<_{lp}) \subseteq legal_{\mathbb{S}}$.

Note that, for a legal linearization, we require that *every* linear extension of $<_{lp}$ yields a legal history under the linearization function lp. Of course, if $\longrightarrow_{hb}$ were total, this would reduce to essentially the standard notion of linearization point, and thus our proof technique is a generalization of a standard technique.

The existence of a legal linearization is sufficient to prove causal linearizability of the C11 execution.

Theorem 16 (Legal linearizations guarantee causal linearizability). *If there is a legal linearization lp, then D is causally linearizable w.r.t.* $\mathbb{S}$.

The key difficulty in using Theorem 16 is showing that a given linearization function is legal. To this end, we extend the standard simulation method to prove legality of a linearization function [16].

In the usual setting, a simulation relation relates states of the implementation to states of the specificiation, and this relation encodes an induction hypothesis for an induction on the executions of the specification. In our current setting, the simulation relation (which we denote ρ, below) relates *sets* of low-level actions to abstract states. The simulation relation encodes an induction hypothesis for an induction on the linear extensions of the hb-relation. Thus, at each stage of the induction we can assume $\rho(Z, \gamma)$ for some set of events $Z \subseteq D$ and state $\gamma \in \Gamma$, where Z is downwards-closed with respect to the hb order. The set Z is the set of low-level actions already considered by the induction. In order to be a simulation, the relation ρ must satisfy the conditions given in the following definition.

Definition 17 (hb-simulation). *Suppose lp is a linearization. An hb-*simulation *is a relation* $\rho \subseteq 2^D \times \Gamma$ *such that:*

1. $\rho(\emptyset, init)$, *and* *(initialisation)*
2. *for all* $Z \subseteq D$, *events* $e \in D \backslash Z$ *and* $\gamma \in \Gamma$, *if* $\forall e' \in D.\ e' \longrightarrow_{hb} e \Rightarrow e' \in Z$ *and* $\rho(Z, \gamma)$ *then*

[6] Thus, each low-level event can linearize at most one action of the specification.

(a) if $e \notin \textbf{ran}\, lp$*, then* $\rho(Z \cup \{e\}, \gamma)$*, and* *(stutter step)*
(b) if $e = lp(i)$ *for some* $i \in D \cap Inv$*,*
then, letting r *be the matching response of* i *in* D*,* $(\gamma, (i, r), \gamma') \in \tau$ *and* $\rho(Z \cup \{e\}, \gamma')$ *for some* $\gamma' \in \Gamma$*.* *(linearization step)*

Condition 1 ensures that the initial states match up: at the concrete level this is the empty set and at the abstract level, this is the initial state. The induction considers the low-level actions in *hb* order, the low-level action under consideration must be an element of $D \backslash Z$ such that all its *hb* predecessors are already in Z. For each such event e, there are two possibilities: either e is a stutter step (in which case the abstract state is unchanged), or e linearizes the operation invoked by i (in which case the abstract system takes a step). In either case, the event e is added to the set Z, and we must show that the simulation relation is preserved.

The existence of an *hb*-simulation guarantees that *lp* is a legal linearization. This fact is captured by the next theorem.

Theorem 18 (*hb*-simulation guarantees legal linearization). *If lp is a linearization and* ρ *is an hb-simulation with respect to lp, then lp is a legal linearization.*

Thus, if we can exhibit a linearization *lp* and an *hb*-simulation ρ, then $\mathbb{D}$ is causally linearizability w.r.t. $\mathbb{S}$.

6.2 Case Study: The Treiber Stack

We now describe a linearization function *lp* and an *hb*-simulation relation ρ, demonstrating causal linearizability of the Treiber stack. We fix some arbitrary C11 execution $\mathbb{D} = (D, sb, rf, mo)$ that contains an instance of the Treiber stack. That is, the invocations in $\mathbb{D}$ are the stack invocations, and the responses are the stack responses (as given in Sect. 3). Furthermore, the low-level memory operations between these invocations and responses are generated by executions of the operations of the Treiber stack (Algorithm 1). As before, we write $e \longrightarrow_{hb} e'$ when $(e, e') \in hb_{\mathbb{D}}$.

The linearization function *lp* for the Treiber stack is completely standard: referring to Algorithm 1 on page 4, each operation is linearized at the unique update operation generated by the unique successful CAS at line 9 (for pushes) or line 16 (for pops).

The main component of our simulation relation ρ guarantees correctness of the *data representation*, i.e., the sequence of values formed by following next pointers starting with & *Top* forms an appropriate stack, and we focus on this aspect of the relation. As is typical with verifications of shared-memory algorithms, there are various other properties that would need to be considered in a full proof.

In a sequentially consistent setting, the data representation can easily be obtained from the state (which maps locations to values). However, for C11 executions calculating the data representation requires a bit more work. In what

follows, we define various functions that depend on a set Z of events, representing the current stage of the induction.

We define the *latest write* in Z to a location x as

$$latest_Z(x) = max(mo\downharpoonright_{(Z \cap Loc(x))})$$

and the *current value* of a location x in some set Z as $cval_Z(x) = wval(latest_Z(x))$, which is the value written by the last write to x in modification order. It is now straightforward to construct the sequence of values corresponding to a location as $stackOf_Z(x) = v \cdot stackOf_Z(y)$, where $v = cval_Z(x.val)$ and $y = cval_Z(x.nxt)$.

Now, assuming that s is a state of the sequential stack, our simulation requires:

$$stackOf_Z(cval_Z(\&Top)) = s \tag{1}$$

Further, we require that all modifications of $\&Top$ are totally ordered by hb:

$$\forall m, m' \in Z \cap Mod(\&Top).\ m \longrightarrow_{hb} m' \vee m' \longrightarrow_{hb} m \tag{2}$$

to ensure that any new read considered by the induction sees the most recent version of $\&Top$.

In what follows, we illustrate how to verify the proof obligations given in Definition 17, for the case where the new event e is a linearization point. Let e be an update operation that is generated by the CAS at line 9 of the push operation in Algorithm 1. The first step is to prove that every modification of $\&Top$ in Z is happens-before the update event e. Formally,

$$\forall m \in Z \cap Mod \cap Loc(\&Top).\ m \longrightarrow_{hb} e \tag{3}$$

Proving this formally is somewhat involved, but the essential reason is as follows. Note that there is an acquiring read r to $\&Top$ executed at line 7 of e's operation and sb-prior to e. r reads from some releasing update u. Thus, by Property 2, and the fact the hb contains sb, e is happens after u, and all prior updates. If there were some update u' of $\&Top$ such that $(u', e) \notin hb$, then $(u', u) \notin hb$ so by Property 2, $u \longrightarrow_{hb} u'$. But it can be shown in this case that the CAS that generated e could not have succeeded, because u' constitutes an update intervening between r and e. Therefore, there can be no such u'.

Property 3 makes it straightforward to verify that Condition 2b of Definition 17 is satisfied. To see this, note that every linearization point of every operation is a modification of $\&\ Top$. Thus, if (i', r') is some operation such that $lp(i') \in Z$ (so that this operation has already been linearized) then $lp(i') \longrightarrow_{hb} e$.

Using Property 3 it is easy to see that both Propertys 1 and 2 are preserved. We show by contradiction that $latest_{Z'}(\&Top) = e$. Otherwise, we have $(e, latest_{Z'}(\&Top)) \in mo$. Therefore $(latest_{Z'}(\&Top), e) \notin hb$, but $latest_{Z'}(\&Top)$ is a modification operation, so this contradicts Property 3.

It follows from $latest_{Z'}(\&Top) = e$ that $stackOf(cval_{Z'}) = stackOf(wval(e))$. Given this, it is straightforward to show that Property 1 is preserved. This step of

the proof relies on certain simple properties of push operations. Specifically, we need to show that the current value of the *val* field of the node being added to the stack (formally, $cval_Z((wval(e)).nxt)$) is the value passed to the push operation; and that the current value of the *nxt* field (formally, $cval_Z((wval(e)).nxt)$) is the current value of $\&Top$ when the successful CAS occurs. These properties can be proved using the model of dynamic memory given in Sect. 5.

7 A Synchronisation Pitfall

We now describe an important observation regarding failure of causal linearizability of read-only operations caused by weak memory effects. The issue can be explained using our abstract notion of an execution structure, however, a solution to the problem is not naturally available in C11 with only release-acquire annotations. Note that this observation does not indicate that our definition of causal linearizability is too strong, but rather that release-acquire annotations cannot guarantee the communication from a read-only operation to a writing operation necessary for compositionality.

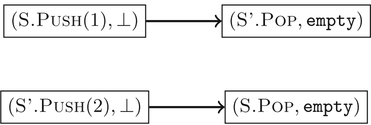


Fig. 3. Read-only operations without communication (not compositional)

Consider the Treiber Stack in Algorithm 1 that returns empty instead of spinning; namely where the inner loop (lines 12–14) is replaced by code block

top $:=^{\mathsf{A}}$ Top ; **if** top = null **then return** `empty`

Such an implementation could produce executions such as the one in Fig. 3 which, like the examples in Sect. 3, is not compositional. Recovering compositionality requires one to introduce additional communication edges from the pops returning empty to the corresponding push operations. In the C11 memory model, these correspond to "from-read" anti-dependencies from a read to a write overwriting the value read. However, release-acquire synchronisation is not adequate for promoting from-read order in the memory to happens-before.

One fix would be to disallow read-only operations, e.g., by introducing a release-acquire CAS operation on a special variable that always succeeds at the start of each operation. However, such a fix is somewhat unnatural. Another would be to use C11's SC annotations, which can induce synchronisation across from-read edges. However, the precise meaning of these annotations is still a topic of active research [7,30].

8 Conclusion and Related Work

We have presented *causal linearizability*, a new correctness condition for objects implemented in weak-memory models, that generalises linearizability and addresses the important problem of compositionality. Our condition is not tied to a particular memory model, but can be readily applied to memory models such as C11, that feature a happens-before relation. We have presented a proof method for verifying causal linearizability. We emphasise that our proof method can be applied directly to a standard axiomatic memory model. Unlike other recent proposals [15,25], we model C11's relaxed accesses without needing to prohibit their problematic dependency cycles (so called "load-buffering" cycles). Although causal linearizability has been presented as a condition for concurrent objects, it is possible to extend this condition to cover, for example, transactional memory.

Causal linearizability is closely related to *causal hb-linearizability* defined in [18], which is a causal relaxation of linearizability that uses specifications strengthened with a happens-before relation. The compositionality result there requires that either a specification is commuting or that a client is unobstructive (does not introduce too much synchronisation). Our result is more general as we place no such restriction on the object or the client. In previous work (see also Sect. 1), we have defined a correctness condition that is only compositional when the client satisfies certain constraints [11]; in contrast, the treatment in this paper achieves a full decoupling between the client and object. Furthermore, that condition is only defined when the underlying memory model is given operationally, rather than axiomatically like C11. Early attempts, targeting TSO architectures, used totally ordered histories but allowed the response of an operation to be moved to a corresponding "flush" event [10,17,21,39]. Others have considered the effects of linearizability in the context of a client abstraction. This includes a correctness condition for C11 that is strictly stronger than linearizability under SC [6]. Although we have applied causal linearizability to C11, causal linearizability itself is more general as it can be applied to any weak memory model with a happens-before relation. Causal consistency [4] is a related condition, aimed at shared-memory and data-stores, which has no notion of precedence order and is not compositional.

There exists a rich body of work on the semantics of weak memory models, including semantics for the C11 memory model [3,5,7,8,26,30,35]. This has been used as a basis for program logics [14,15,22,25,29,38] and given rise to automated tools for analysis of weak memory programs [1,2,27,37]. These logics and associated verification tools however, are typically not designed to reason about simulation and refinement as is essential for proofs of (causal) linearizability of concurrent data structures [16]. There are several existing automated techniques for checking (classical) linearizability [9,19,33,41] that use simulation-based techniques. We anticipate that such techniques could be extended to verify hb-simulation and causal linearizability, however, leave such extensions as future work.

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. Acta Inform. **54**(8), 789–818 (2017)
2. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: Context-bounded analysis for POWER. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 56–74. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_4
3. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: a tutorial. IEEE Comput. **29**(12), 66–76 (1996)
4. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. Distrib. Comput. **9**(1), 37–49 (1995)
5. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (2014)
6. Batty, M., Dodds, M., Gotsman, A.: Library abstraction for C/C++ concurrency. In: Giacobazzi, R., Cousot, R. (eds.) POPL, pp. 235–248. ACM (2013)
7. Batty, M., Donaldson, A.F., Wickerson, J.: Overhauling SC atomics in C11 and OpenCL. In: POPL, pp. 634–648. ACM (2016)
8. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Ball, T., Sagiv, M. (eds.) POPL, pp. 55–66. ACM (2011)
9. Bouajjani, A., Emmi, M., Enea, C., Mutluergil, S.O.: Proving linearizability using forward simulations. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 542–563. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_28
10. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent library correctness on the TSO memory model. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 87–107. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_5
11. Doherty, S., Derrick, J.: Linearizability and causality. In: De Nicola, R., Kühn, E. (eds.) SEFM 2016. LNCS, vol. 9763, pp. 45–60. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41591-8_4
12. Doherty, S., Derrick, J., Dongol, B., Wehrheim, H.: Causal linearizability: compositionality for partially ordered executions. CoRR abs/1802.01866 (2018)
13. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30232-2_7
14. Doko, M., Vafeiadis, V.: A program logic for C11 memory fences. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 413–430. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_20
15. Doko, M., Vafeiadis, V.: Tackling real-life relaxed concurrency with FSL++. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 448–475. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_17
16. Dongol, B., Derrick, J.: Verifying linearisability: a comparative survey. ACM Comput. Surv. **48**(2), 19:1–19:43 (2015)
17. Dongol, B., Derrick, J., Smith, G.: Reasoning algebraically about refinement on TSO architectures. In: Ciobanu, G., Méry, D. (eds.) ICTAC 2014. LNCS, vol. 8687, pp. 151–168. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10882-7_10

18. Dongol, B., Jagadeesan, R., Riely, J., Armstrong, A.: On abstraction and compositionality for weak-memory linearisability. Verification, Model Checking, and Abstract Interpretation. LNCS, vol. 10747, pp. 183–204. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73721-8_9
19. Doolan, P., Smith, G., Zhang, C., Krishnan, P.: Improving the scalability of automatic linearizability checking in SPIN. In: Duan, Z., Ong, L. (eds.) ICFEM 2017. LNCS, vol. 10610, pp. 105–121. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68690-5_7
20. Filipovic, I., O'Hearn, P.W., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theor. Comput. Sci. **411**(51–52), 4379–4398 (2010)
21. Gotsman, A., Musuvathi, M., Yang, H.: Show no weakness: sequentially consistent specifications of TSO libraries. In: Aguilera, M.K. (ed.) DISC 2012. LNCS, vol. 7611, pp. 31–45. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33651-5_3
22. He, M., Vafeiadis, V., Qin, S., Ferreira, J.F.: Reasoning about fences and relaxed atomics. In: PDP, pp. 520–527. IEEE Computer Society (2016)
23. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, Burlington (2008)
24. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM TOPLAS **12**(3), 463–492 (1990)
25. Kaiser, J., Dang, H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: reasoning about release-acquire consistency in iris. In: ECOOP. LIPIcs, vol. 74, pp. 17:1–17:29. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
26. Kang, J., Hur, C., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: Castagna, G., Gordon, A.D. (eds.) POPL, pp. 175–189. ACM (2017)
27. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. PACMPL **2**(POPL), 17:1–17:32 (2018)
28. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: Bodík, R., Majumdar, R. (eds.) POPL, pp. 649–662. ACM (2016)
29. Lahav, O., Vafeiadis, V.: Owicki-Gries reasoning for weak memory models. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 311–323. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_25
30. Lahav, O., Vafeiadis, V., Kang, J., Hur, C., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI, pp. 618–632. ACM (2017)
31. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9), 690–691 (1979)
32. Lamport, L.: On interprocess communication. Part I: basic formalism. Distrib. Comput. **1**(2), 77–85 (1986)
33. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 321–337. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05089-3_21
34. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL, pp. 378–391. ACM (2005)
35. Nienhuis, K., Memarian, K., Sewell, P.: An operational semantics for C/C++11 concurrency. In: Visser, E., Smaragdakis, Y. (eds.) OOPSLA, pp. 111–128. ACM (2016)
36. Schellhorn, G., Derrick, J., Wehrheim, H.: A sound and complete proof technique for linearizability of concurrent data structures. ACM Trans. Comput. Log. **15**(4), 31:1–31:37 (2014)

37. Summers, A.J., Müller, P.: Automating deductive verification for weak-memory programs. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 190–209. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_11
38. Svendsen, K., Pichon-Pharabod, J., Doko, M., Lahav, O., Vafeiadis, V.: A separation logic for a promising semantics. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 357–384. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_13
39. Travkin, O., Wehrheim, H.: Handling TSO in mechanized linearizability proofs. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 132–147. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13338-6_11
40. Treiber, R.K.: Systems programming: coping with parallelism. Technical report, RJ 5118, IBM Almaden Res. Ctr. (1986)
41. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_40

Security Protocol Analysis in Context: Computing Minimal Executions Using SMT and CPSA

Daniel J. Dougherty[1(✉)], Joshua D. Guttman[1,2], and John D. Ramsdell[2]

[1] Worcester Polytechnic Institute, Worcester, MA, USA
dd@wpi.edu
[2] The MITRE Corporation, Bedford, MA, USA

Abstract. Cryptographic protocols are used in different environments, but existing methods for protocol analysis focus only on the protocols, without being sensitive to assumptions about their environments.

LPA is a tool that analyzes protocols in context. LPA uses two programs, cooperating with each other: CPSA, a well-known system for protocol analysis, and Razor, a model-finder based on SMT technology. Our analysis follows the enrich-by-need paradigm, in which models of protocol execution are generated and examined.

The choice of which models to generate is important, and we motivate and evaluate LPA's strategy of building *minimal* models. "Minimality" can be defined with respect to either of two preorders, namely the homomorphism preorder and the embedding preorder (i.e. the preorder of injective homomorphisms); we discuss the merits of each. Our main technical contributions are algorithms for building homomorphism-minimal models and for generating a set-of-support for the models of a theory, in each case by scripting interactions with an SMT solver.

1 Introduction

Cryptographic protocol analysis is well-developed. Many tools and rigorous techniques can determine what confidentiality, authentication (e.g. [5,11,16,35]), and indistinguishability properties (e.g. [6,8,9]) protocols satisfy.

However, what goals a protocol *needs* to achieve depends on the applications that use it. The applications require certain security functionality; a protocol is acceptable if it achieves at least what that functionality relies on. Often, an attack shows that a protocol ensures less than an application needed. For instance, the TLS resumption attacks [37], cf. [4,41] show that the protocol did not allow the server application to distinguish unauthenticated input at the beginning of a data stream from subsequent authenticated input. This may lead to erroneous authorization decisions.

Conversely, a protocol may be good enough for an application because of *environmental assumptions* the application ensures. For instance, some protocols

This work was partially support by the US NSF under Grant No. 1714431.

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 130–150, 2018.
https://doi.org/10.1007/978-3-319-98938-9_8

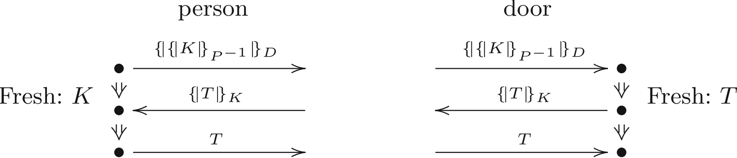


Fig. 1. DoorSEP protocol

fail if the same long-term key is ever used by a principal when playing the server role and also when playing a client role. However, some applications ensure that no server ever executes the protocol in the client role at all. This policy would ensure that an otherwise weak protocol reliably supports the application's needs.

Logical Protocol Analysis is our term for combining a protocol analyzer with these additional concerns, which we analyze via model finding. Our goal is to analyze cryptographic protocols that include trust axioms that cannot be stated using the typical input to a protocol analyzer. We will carry this idea out using the model finder Razor [42] and CPSA, a specialized protocol analysis tool [20,36].

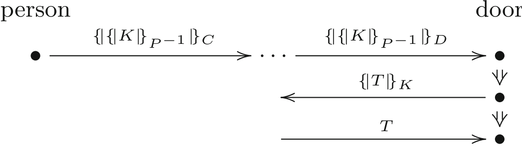
Flawed protocols are often deployed before the flaws are understood, and embedded in widely used devices. Such protocols can still achieve desired security goals when used in a restricted context. If the context can be modeled using environmental assumptions and other trust axioms, Logical Protocol Analysis can be used to discover whether the goals are met in the actual context of use.

An Example: DoorSEP. As a motivating scenario consider the Door Simple Example Protocol (DoorSEP), derived from an expository protocol [7] that was designed to have a weakness. Despite this, the protocol achieves the needs of an application, given a trust assumption. Section 4.1 has more detail.

Imagine a door D that is equipped with a badge reader, and a person P equipped with a badge. When the person swipes the badge, the protocol executes. Principals such as doors or persons are identified by the public parts of their key pairs, with D^{-1} and P^{-1} being the corresponding private keys. We write $\{\!|M|\!\}_K$ for the encryption of message M with key K. We represent digital signatures $\{\!|M|\!\}_{P^{-1}}$ as if they were the result of encrypting with P's private key.

P initiates the exchange by creating a fresh symmetric key K, signing it, and sending it to the door D encrypted with the door's public key. D extracts the symmetric key after checking the signature, freshly generates a token T, and sends it—encrypted with the symmetric key—back to P. P demonstrates they are authorized to enter by decrypting the token and sending it as plaintext to the door. The two roles of DoorSEP are shown in Fig. 1, where each vertical column displays the behavior of one of the roles.

CPSA finds an undesirable execution of DoorSEP. Assume the person's private key P^{-1} is uncompromised and the door has received the token it sent out. Then CPSA finds that P freshly created the symmetric key K. However, nothing ensures that the person meant to open door D. If P ever initiates a run with



Uncompromised: P Fresh: K, T

Fig. 2. DoorSEP weakness

a compromised door C, the adversary can perform a man-in-the-middle attack, decrypting using the compromised key C^{-1} and re-encrypting with D's public key, as elided in the $\cdots$ in Fig. 2. Thus, without additional assumptions, the door cannot authenticate the person requesting entry.

But possibly we can trust the person to swipe her badge only in front of doors our organization controls. And we can ensure that our doors have uncompromised private keys. If so, then the adversary cannot exercise the flaw. We regard this as a *trust assumption*, and we can express it as an axiom:

Trust Assumption 1. If an uncompromised signing key P^{-1} is used to prepare an instance of the first DoorSEP message, then its owning principal has ensured that the selected door D has an uncompromised private key.

The responsibility for ensuring the truth of this axiom may be split between P and the organization controlling D. P makes sure to swipe her badge only at legitimate doors of the organization's buildings. The organization maintains a security posture that protects the corresponding private keys.

Is DoorSEP good enough, assuming the trust axiom? To analyze DoorSEP under trust assumption 1, we use a model finder, namely *Razor* [42]. We provide it a theory leading to a model containing the man-in-the-middle attack. We then add the trust axiom above. The axiom entails that the adversary cannot decrypt the message sent by the P.

The generated model is then given to CPSA, which infers that the door can decrypt the person's message only if $C = D$, i.e. if P intended it D. Thus, the protocol does its job; namely, ensuring that the door opens only when an authorized person requests it to open.

1.1 Protocols and Theories

Security conclusions require protocol analysis combined with other properties, which we will assume are given axiomatically by a theory $\mathcal{G}$. We also regard a protocol Π as determining an axiomatic theory $Th(\Pi)$, namely the theory of Π's executions, as Π runs possibly in the presence of a malicious adversary. Thus, we would like to understand the joint models of $\mathcal{G} \cup Th(\Pi)$, where of course these theories may share vocabulary. In the DoorSEP case, $\mathcal{G}$ is the trust axiom.

The models of $\mathcal{G} \cup Th(\Pi)$ are runs of DoorSEP in which the doors and people act as assumed in $\mathcal{G}$.

Enrich-by-Need. Indeed, our approach is to construct *minimal models* in a *homomorphism order*. We refer to these minimal models as *shapes* [20]. The shapes show all of the minimal, essentially different things that can happen subject to $\mathcal{G} \cup Th(\Pi)$: every execution contains instances—meaning homomorphic images—of the shapes. This is useful to the security analyst who can inspect the minimal models and appraise whether they are compatible with his needs. The analyst can do this even without being able to explicitly state the key security goals. In the case in which $\mathcal{G} = \emptyset$, so that only $Th(\Pi)$ matters, generating these shapes is the central functionality of CPSA [36].

We call this approach to security analysis *enrich-by-need*, since we build homomorphism-minimal models by rising stepwise in the homomorphism order, gradually generating them all. CPSA does so using a "authentication test" method, which yields a compact, uniform way to generate the set of minimal models of the protocol theory [20,27].

Indeed, when the set of shapes is finite, we can summarize them in a formula, the disjunction of the *diagrams* of each. We regard this as the conclusion of an implication; the diagram of the starting scenario is the hypothesis. This *shape analysis sentence* is a strongest security goal achieved by the protocol that has the hypothesis chosen [21,34]. Lemma 8 justifies this idea.

We extend CPSA here to cooperate with another tool to provide models of the whole theory $\mathcal{G} \cup Th(\Pi)$. We effectively split $Th(\Pi)$ into two parts, a hard part T_h and an easy part T_e. Only CPSA will handle the hard part.

We use a general-purpose model-finder, *Razor* [42] to look for minimal models of $\mathcal{G} \cup T_e$ that extend a fragment of a model. When the resulting model $\mathbb{A}$ contains additional behavior of Π, we return to CPSA to handle the hard part T_h, enriching $\mathbb{A}$ with some possible executions. We then return these extensions to Razor. If this process terminates, we have a minimal joint model. By iterating our search, we obtain a covering set of minimal joint models. Razor, in turn, is built as a wrapper around a Satisfiability Modulo Theories (SMT) solver, specifically Z3 [12].

Contributions. We have two goals. First, we define and justify the methods that the new Razor uses to drive Z3 to generate homomorphism-minimal models of a given theory. These homomorphisms are not necessarily embeddings; that is, a homomorphism to construct may map distinct values in its source model to the same value in its target model. To begin with, we need a method to construct, from a model $\mathbb{A}$, a set of sentences $homFrom_{\mathbb{A}}$, true in precisely those models $\mathbb{B}$ such that there is a homomorphism from $\mathbb{A}$ to $\mathbb{B}$. We also need a method to construct, from a model $\mathbb{A}$, a set of sentences $homTo_{\mathbb{A}}$, true in precisely those models $\mathbb{B}$ such that there is a homomorphism from $\mathbb{B}$ to $\mathbb{A}$. We show how to use these two resources to compute a set of minimal models that covers all of the models; this method is codified in Razor.

Second, we develop a particular architecture for coordinating Razor and CPSA. In this architecture, Razor handles all aspects of $\mathcal{G} \cup Th(\Pi)$ *except* that

it does not enrich a fragmentary execution of Π to obtain its shapes, i.e. the minimal executions that are its images. Instead, we generate an input to CPSA that contains the substructure $\mathbb{A}_0$ containing only protocol behavior. CPSA computes the shapes and extracts the strongest security goal that applies to $\mathbb{A}_0$. It returns this additional information to Razor, which then iterates. We call this cooperative architecture LPA for *Logical Protocol Analysis*.

Structure of the Paper. In Sect. 2 we fix some preliminary definitions and notation; we introduce the two existing tools that coordinate to make LPA in Sect. 3. In Sect. 4 we describe LPA itself and how it is used to analyze the DoorSEP protocol. Section 5 is a development of some of the underlying theory of using SMT solving to compute and present models, with an emphasis on the question: *which models should be presented to the user?* We end with conclusions and a discussion of future work. Some proofs have been omitted, and some discussion condensed, for lack of space; see [14] for a fuller treatment.

Related Work. Model-finding is an active area of investigation [3,10,29,38,43]. But existing model-finders compute an essentially random set of models. Close in spirit to our goals and techniques are lightweight formal methods tools such as Alloy [26] and Margrave [31]. Aluminum [32] supports exploration by returning minimal models: it instruments the model-finding engine of Alloy.

Logic programming languages produce single, *least* models as a consequence of their semantics; this is not a notion of minimality based on homomorphisms, and is traditionally tied to Horn-clause theories. Generalizations of minimality for non-Horn theories have already been used in specifying the semantics of disjunctive logic programming [28] and in non-monotonic reasoning, especially circumscription [39].

Our previous work on a Cryptographic Protocol Programming Language [22, 24] led to a programming language that would allow protocol actions to be controlled by a trust management policy.

The Tamarin prover [30] can limit the context in which a protocol is to be analyzed by restricting its analysis to a user-specified subset of all protocol traces. In contrast, our primary interests lie in *enriching* the context in which analysis is done and in generating principled output instances. There was also related work in the applied π-calculus [18,19]. Protocol analysis sometimes builds in environmental assumptions in a security goal hypothesis, by assuming that some keys are uncompromised, or that some principal names are unequal. However, the focus of research has been on the pure problem of determining the security properties of protocols in isolation.

2 Foundations

2.1 Models and Homomorphisms

In this chapter we present some of the foundations of model-finding, focusing on the use of an SMT solver. In broadest terms, model-finding is the following task: given a logical theory $\mathcal{T}$, produce one or more (finite) models of $\mathcal{T}$.

Of course a typical satisfiable theory will have many models. Special emphasis is given in this paper to the question of *which models should be presented to the user?* One answer—embodied in the LPA tool—is based on the fundamental notion of *homomorphism* between models, with a focus on models that are *minimal* (see Sect. 5) in the pre-order determined by homomorphism.

Fix a signature Σ. A *model* $\mathbb{A}$ for signature Σ is defined as usual: a collection of sets interpreting the sorts of Σ, and a collection of functions and relations interpreting the function and relation symbols of Σ. In this paper we work with finite models exclusively.

Definition 1. *Let* $\mathbb{A}$ *and* $\mathbb{B}$ *be* Σ*-models. A* homomorphism *from* $\mathbb{A}$ *to* $\mathbb{B}$ *is a sort-indexed family of maps such that*

1. $\mathbb{A} \models f[a_1, \ldots, a_n] = a$ *implies* $\mathbb{B} \models f[h(a_1), \ldots, h(a_n)] = h(a)$ *and*
2. $\mathbb{A} \models R[a_1, \ldots, a_n]$ *implies* $\mathbb{B} \models R[h(a_1), \ldots, h(a_n)]$.

Write $\mathbb{A} \precsim \mathbb{B}$ if there is a homomorphism $h : \mathbb{A} \to \mathbb{B}$, and write $\mathbb{A} \approx \mathbb{B}$ if $\mathbb{A} \precsim \mathbb{B}$ and $\mathbb{B} \precsim \mathbb{A}$. Write $\mathbb{A} \precsim^i \mathbb{B}$ if there is an injective homomorphism $h : \mathbb{A} \to \mathbb{B}$, and write $\mathbb{A} \approx^i \mathbb{B}$ if $\mathbb{A} \precsim^i \mathbb{B}$ and $\mathbb{B} \precsim^i \mathbb{A}$. We will sometimes use the phrase "hom-cone of $\mathbb{A}$" to refer to the set of models $\mathbb{B}$ for which there is a homomorphism $h : \mathbb{A} \to \mathbb{B}$.

Definition 2. *Let* $\mathcal{M}$ *be a class of models. A model* $\mathbb{M} \in \mathcal{M}$ *is a*-minimal *for* $\mathcal{M}$ *if whenever* $\mathbb{A} \in \mathcal{M}$ *and* $\mathbb{A} \precsim \mathbb{M}$*, we have* $\mathbb{A} \approx \mathbb{M}$*. The definition of i-minimal is similar, using injective homomorphisms. (The modifier "a−" is to suggest "arbitrary".)*

The notion of the *core* of a model is standard [17,25]; it is important for us because cores will give canonical representatives of $\approx$ equivalence classes.

Core are defined in terms of *retractions,* as follows.

Definition 3. *A* retraction $r : \mathbb{A} \to \mathbb{B}$ *is a homomorphism such that there is a homomorphism* $e : \mathbb{B} \to \mathbb{A}$ *with* $r \circ e = \mathsf{id}_{\mathbb{B}}$*.*

A submodel $\mathbb{C}$ *of* $\mathbb{A}$ *is a* core *of* $\mathbb{A}$ *if there is a retraction* $r : \mathbb{A} \to \mathbb{C}$ *but no retract* $r' : \mathbb{A} \to \mathbb{C}'$ *for any proper submodel* $\mathbb{C}'$ *of* $\mathbb{C}$*.*

A model $\mathbb{C}$ *is a* core *if it is a core of itself.*

Definition 4 (PE formula, Geometric theory). *A formula is* positive-existential, *or* PE*, if it is built from atomic formulas (including* true *and* false*) using* $\wedge$*,* $\vee$ *and* $\exists$*. A* geometric *sentence is one of the form*

$$\forall \vec{x}. \quad \alpha(\vec{x}) \to \beta(\vec{x})$$

where α *and* β *are positive-existential.*

Theorem 5. *The following are equivalent, for a formula* $\alpha(\vec{x})$*:*

1. α *is preserved by homomorphism: if* $h : \mathbb{A} \to \mathbb{B}$ *is a homomorphism, and* $\vec{a}$ *is a vector of elements from* $\mathbb{A}$ *such that* $\mathbb{A} \models \alpha[\vec{a}]$*, then* $\mathbb{B} \models \alpha[\vec{ha}]$*.*

2. α is logically equivalent to a PE formula.
3. α is equivalent, in the category $\mathcal{M}_\Sigma$ of finite models, to a PE formula.

Proof. The equivalence of (1) and (2) is a classical result in model theory when considering arbitrary models. The equivalence of (1) and (3) is a deep result of Rossman [40].

The case for geometric logic as a logic of observable properties was made clearly by Abramsky [1]. As detailed in [21], typical security goals for protocols are naturally expressed as geometric sentences. (As is well-known, *any* theory is equisatisfiable with one in conjunctive normal form, by introducing Skolem functions. Such an enrichment of the theory signature is not innocent, however, since it has consequences for the existence of homomorphisms between models.)

It is straightforward to see that when T is geometric, if $\mathbb{A}$ is a model of T then a retraction of $\mathbb{A}$ is a model of T.

Lemma 6. *Let T be a geometric theory, $\mathbb{A} \models T$, and $r : \mathbb{A} \to \mathbb{B}$ a retraction. Then $\mathbb{B} \models T$.*

Definition 7. *If $\mathcal{M}$ is a class of Σ-models and $\mathcal{M}_0 \subseteq \mathcal{M}$ say that $\mathcal{M}_0$ is an* a-set of support for *$\mathcal{M}$ if for all $\mathbb{B} \in \mathcal{M}$, there exists $\mathbb{A} \in \mathcal{M}_0$ with $\mathbb{A} \precsim \mathbb{B}$. Similarly for i-set of support.*

A set of support for a class of models provides a complete "testbed" for entailment of geometric sentences:

Lemma 8. *Let $\sigma \equiv \forall \vec{x}.\ \alpha(\vec{x}) \to \beta(\vec{x})$ be geometric and let $\mathcal{M}$ be a class of models. Let $\mathcal{M}_0$ be an a-set of support for $\{\mathbb{A} \in \mathcal{M} \mid \mathbb{A} \models \exists \vec{x}.\ \alpha(\vec{x})\}$. If every model in $\mathcal{M}_0$ satisfies σ then every model in $\mathcal{M}$ satisfies σ.*

Proof. Let $\mathbb{P} \in \mathcal{M}$ with $\mathbb{P} \models \alpha[\vec{a}]$; we want to show that $\mathbb{P} \models \beta[\vec{a}]$. Let $\mathbb{M} \in \mathcal{M}_0$ with $\mathbb{M} \precsim \mathbb{P}$. Since $\mathbb{M} \models \sigma$, $\mathbb{M} \models \beta[\vec{a}]$. Since β is PE and $\mathbb{M} \precsim \mathbb{P}$, $\mathbb{P} \models \beta[\vec{a}]$.

2.2 Strand Spaces

We can formalize protocol executions as models, as follows. A run of a protocol is viewed as an exchange of messages by a finite set of local sessions of the protocol. Each local session is called a *strand:* a strand is a sequence of nodes n, each of which is a *transmission* or a *reception* of the *message* $msg(n)$ at that node.

A *strand space* Θ is a finite sequence of strands. A message that originates in exactly one strand of Θ is *uniquely originating*, and represents a freshly chosen value. A message is *mentioned* in Θ if it occurs in a strand of Θ, or if it is an asymmetric key, its inverse occurs in a strand of Θ. A message that is mentioned but originates nowhere in Θ is *non-originating*, and often represents an uncompromised key.

A *protocol* Π is a finite set of strands, which are the *roles* of the protocol. A strand s is an *instance* of a role $\rho \in \Pi$, if $s = \alpha(\rho)$, i.e. if s results from ρ by applying a substitution α to parameters in ρ.

Skeletons are fragmentary executions of the regular participants, which factor out adversary behavior. A *skeleton* $\mathbb{K} = (\mathsf{nodes}, \preceq, \mathsf{non}, \mathsf{unique})$ consists of a finite set of regular nodes, a partial ordering on them, a set of values assumed non-originating, and a set of values assumed uniquely originating. These components are designed to code in the aspects of executions that we care about, namely the ordering, and what values are uncompromised ("non") or freshly chosen ("unique").

A skeleton $\mathbb{K}$ is an *execution*, or *realized*, iff for every message received in $\mathbb{K}$, the Dolev-Yao adversary [13] can derive that message with the help of earlier transmissions in $\mathbb{K}$.

Associated with each CPSA protocol Π is a first-order language $\mathcal{L}(\Pi)$ used to specify security goals [21]. The language can be used to exchange information between CPSA and an SMT solver. These mechanisms are described in Sect. 4.

3 Constituent Tools

CPSA. The Cryptographic Protocol Shapes Analyzer [35] (CPSA) can be used to determine if a protocol achieves authentication and secrecy goals. CPSA will—given a protocol Π and a skeleton of interest $\mathbb{K}$—generate all of the minimal, essentially different realized skeletons that are homomorphic images of $\mathbb{K}$. We call these minimal, essentially different skeletons *shapes*, and, although in general there could be infinitely many of them, frequently there are very few of them.

CPSA begins a run with a protocol description and an initial scenario $\mathbb{K}_0$. The initial scenario is a partial description of executions of a protocol. If CPSA terminates, it characterizes all the executions of the protocol consistent with the initial scenario. For example, if it is assumed that one role of a protocol runs to completion, CPSA will determine what other roles must have executed.

Each skeleton $\mathbb{K}$ has a *characteristic sentence* $\sigma_{\mathbb{K}}$ such that, for all $\mathbb{K}'$, $h : \mathbb{K} \to \mathbb{K}'$ (for some homomorphism h) iff $\mathbb{K}' \models \sigma_{\mathbb{K}}$.

Homomorphisms play an essential role in CPSA. At each step in the algorithm, an unrealized skeleton $\mathbb{K}$ is replaced by a set of skeletons $\{\mathbb{K}_1, \ldots, \mathbb{K}_n\}$, called a cohort, by solving an authentication test [23]. The skeletons $\{\mathbb{K}_1, \ldots, \mathbb{K}_n\}$ form an *a-set of support* for the realized skeletons that are homomorphic images of $\mathbb{K}$. That is, if there is an execution (or "realized skeleton") $\mathbb{K}_r$ such that $h : \mathbb{K} \to \mathbb{K}_r$, then there exists some homomorphism $h' : \mathbb{K}_i \to \mathbb{K}_r$ such that $h = h' \circ h_i$.

For an initial scenario $\mathbb{K}_0$, CPSA produces a set of realized skeletons $\{\mathbb{K}_1, \ldots, \mathbb{K}_n\}$ and homomorphisms $h_i : \mathbb{K}_0 \to \mathbb{K}_i$. These are built up by a succession of cohort steps; thus, they remain an a-set of support for the realized skeletons that are homomorphic images of $\mathbb{K}_0$. The set $h_i : \mathbb{K}_0 \to \mathbb{K}_i$—called the *shapes* of this scenario—are a compact way of describing all of the executions compatible with the initial scenario. By Lemma 8, if a geometric sentence σ holds in each shape, then σ holds in every realized skeleton that is an image of $\mathbb{K}_0$.

There is a key geometric sentence that can be extracted from the results of a run of CPSA. A Shape Analysis Sentence (SAS) [34] encodes everything that has been learned about the protocol from a CPSA analysis starting with a given initial scenario. It holds in every realized skeleton of the protocol. A SAS is used to import the results of a CPSA analysis into the SMT solver.

The antecedent of a SAS is a conjunction of atomic formulas that specify the initial scenario $\mathbb{K}_0$. The universally quantified variables are the ones that occur in the antecedent. The conclusion is a disjunction of formulas, one for each shape. The ith disjunct is an existentially quantified conjunction of atomic formulas that describes the mapping h_i and the additions to the antecedent required to specify shape $\mathbb{K}_i$.

Razor. Razor is a general-purpose model-finder: it takes as input an arbitrary first-order theory T and attempts to find finite models of T (CPSA can be viewed as a domain-specific model-finder, working over various theories of strand spaces).

Razor finds models by (i) preprocessing the input theory as described below, (ii) using an off-the-shelf SMT solver, currently Z3, and (iii) postprocessing the results of the solver's output to fulfill certain goals: return *minimal* models by default, allowing the user to explore and augment models, and computing a set-of-support of models for T. Razor can be used in REPL mode or batch mode; only the latter is used as part of LPA (refer to [42] for a fuller description of Razor's REPL mode).

Once the SMT solver has determined that a theory T is satisfiable, and computed—internally—a model for T, the application must extract the model from the solver. But the API mandated by the SMT-Lib Standard (v.2.6) [2] for doing this is quite restricted. The model can be inspected only through certain commands returning the solver's internal representation of values of terms.

This is inconvenient for us, especially since the solver might create only a partial model internally.

To address this, we first ensure that the language we use to communicate with the solver has enough ground terms at each sort to name all elements of a model, by adding fresh constants. Then we can query the solver for the values of the functions and predicates, and build a "basic" model representation

$$\begin{aligned} \text{equations} \quad & c_i = c_j && \text{and} \\ \text{equations} \quad & f\vec{c} = c && \text{and} \\ \text{facts} \quad & R\vec{c} \end{aligned}$$

where the c_i range over the fresh constants. Using standard techniques we then construct from these equations a convergent (terminating and confluent) ground rewrite system, which facilitates working with the models.

4 LPA

This section shows how to use CPSA and Razor to analyze cryptographic protocols in context. Our architecture for LPA is displayed in Fig. 3. An analysis begins with a CPSA protocol Π and an initial theory T_0. The initial theory contains a specification of the trust policy and a description of the initial scenario of the protocol as a collection of sentences in $\mathcal{L}^+(\Pi)$, an extension of $\mathcal{L}(\Pi)$.

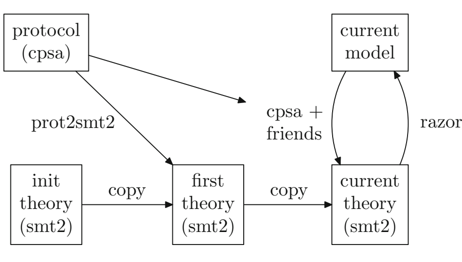


Fig. 3. LPA architecture

The program `prot2smt2` uses protocol Π to generate a set of geometric axioms $Th(\Pi)$ [14, Sect. 2.1.5]. These axioms allow Razor to produce models from which skeletons can be extracted. For example, an axiom about the transitivity of node orderings allows Razor to compute the partial ordering of the nodes. Other axioms ensure that a uniquely originating value is received only after it is transmitted and that the double inverse of each asymmetric key is equal to itself.

The initial theory is appended to $Th(\Pi)$ to form the first theory T_1 to be analyzed by Razor. A skeleton is extracted from each model. If the skeleton is realized, the model describes the impact of the trust policy on complete executions of the protocol. If the skeleton is not realized, it is used as the initial scenario for CPSA. The results of CPSA is turned into a SAS (shape analysis sentence, cf. Sect. 3) and added to the current theory for further analysis. The process is repeated until all of the extracted skeletons are realized.

4.1 Analyzing the Door Simple Example Protocol

We now expand on the analysis of the DoorSEP protocol introduced in Sect. 1. In this protocol, a person begins by generating a fresh symmetric key, signing it, and then encrypting the result using the door's public key. If the door accepts the first message, it responds by freshly generating a token and uses the symmetric key to encrypt it. If the door receives the token back unencrypted, the door concludes the person that generated the key is at the door and opens.

The initial theory specifies Trust Assumption 1 and the fact that the door is open. To assert the door is open, one asserts there is a strand that is a full length instance of the door role. We further assert that the person's private key is uncompromised.

Recall the diagram in Fig. 3 to visualize the analysis process. After appending the initial theory to the protocol axioms $Th(\Pi)$, Razor finds model $\mathcal{M}_0$. As expected, model $\mathcal{M}_0$ specifies a full length door strand in which the person's private key is uncompromised and other facts such as the fact that double inverse of the model's asymmetric keys are equal to themselves.

At this stage, we have a model that characterizes an unrealized skeleton, and we would like to use CPSA to find out what else must have happened. The shape produced by CPSA is displayed in Fig. 2. The shape shows the lack of mutual authentication built into this flawed protocol.

The next step in the analysis makes use of the trust axiom. The result of the CPSA analysis is transformed into a SAS. The antecedent specifies the initial scenario described by the first model. The consequence specifies what else must be added to make the initial scenario into the complete execution shown in Fig. 2.

When the SAS is added to the current theory, Razor finds one model $\mathcal{M}_1$. The skeleton extracted from this model is very similar to the shape in Fig. 2 with one crucial difference: the key D' is uncompromised. Razor applied the trust axiom. The skeleton extracted from $\mathcal{M}_1$ is unrealized, so CPSA can make a contribution. It finds a SAS that extends the length of the person strand to full length and equates D and D'. The addition of this SAS produces model $\mathcal{M}_2$ that characterizes a realized skeleton with full agreement between the door and person strands. Because the skeleton is realized, CPSA has nothing more to contribute and the analysis terminates.

5 Minimality, Cores, and Set-of-Support

In this section we explore the question *which models should we compute and show to the user of a model-finding tool?* Our proposal, motivated by Lemma 8 and implemented by LPA, is: *compute a set-of-support for the input theory comprised of minimal models.* As we have observed there are two natural notions of minimality; we point out some theoretical differences between them. Most importantly, we present algorithms for computing minimal models and sets-of-support: these involve programming against the functionality of SMT solvers.

5.1 Comparing *i*-Minimal and *a*-Minimal

One way to think about a-minimality of a model $\mathbb{M}$ is that if any atomic fact of $\mathbb{M}$ is removed, the resulting model would no longer be a model of the theory at hand. In particular, since equality is an atomic predicate, if two terms denote—unnecessarily—the same model-element, this is a failure of a-minimality.

Neither of i-minimality or a-minimality implies the other.

Example 9.

- Let T be the single sentence $\exists x.P(x) \wedge \exists x.Q(x)$, and let $\mathbb{A}$ have one element a with $\mathbb{A} \models P[a] \wedge Q[a]$.
  Then $\mathbb{A}$ is i-minimal but not a-minimal: the model $\mathbb{B}$ with two elements a_1 and a_2 such that $\mathbb{B} \models P[a_1] \wedge Q[a_2]$ is strictly below $\mathbb{A}$ in the $\precsim$ preorder. ($\mathbb{B}$ is a-minimal for T.)
- Let T be $\exists x.P(x)$ and let $\mathbb{A}$ have two elements a_1 and a_2 with $\mathbb{A} \models P[a_1]$ and $\mathbb{A} \models P[a_2]$. Then $\mathbb{A}$ is a-minimal but is not i-minimal: the induced model determined by a_1 is a model of T.

However, an a-minimal model that is a core *will* be i-minimal.

Lemma 10. *If $\mathbb{A}$ is a-minimal for T and is a core, then $\mathbb{A}$ is i-minimal for T.*

Proof. Suppose $\mathbb{B}$ is a model of T and $j : \mathbb{B} \to \mathbb{A}$ is injective. Since $\mathbb{A}$ is a-minimal, there is a homomorphism $h : \mathbb{A} \to \mathbb{B}$. The composition $j \circ h$ is an endomorphism of $\mathbb{A}$. Since $\mathbb{A}$ is a core this map is injective, so h is injective, and $\mathbb{A} \approx^i \mathbb{B}$.

We should observe that for a given theory there might be no finite a-minimal models at all. An example is the theory with one unary function and no axioms. The initial (hence unique minimal) model of this theory is the natural numbers. Another way to put this is: the $\precnsim$ preorder is not well-founded in general.

On the other hand, we will typically add axioms to a theory to ensure that there is an upper bound on the size of its models. In such a case there will be only finitely many models of T, and the $\precnsim$ preorder will be well-founded. This is the key to the termination of many of the algorithms in this section. (There will always be a-minimal models for theories T that are bounded in this way.)

Lemma 11. *Let T be a theory with only finitely many models. Then the $\precnsim$ and $\precnsim^i$ preorders on models of T are well-founded.*

Proof. Suppose for the sake of contradiction that we have an infinite descending chain $\ldots \precnsim \mathbb{M}_2 \precnsim \mathbb{M}_1 \precnsim \mathbb{M}_0$ of strict homomorphisms. Then we have $\mathbb{M}_{i+k} \precnsim \mathbb{M}_i$ for any $k \geq 0$. Since T has finitely many models, we eventually get i and $k \geq 0$ with $\mathbb{M}_{i+k+1}$ isomorphic to $\mathbb{M}_i$. So $\mathbb{M}_{i+k+1} \precnsim \mathbb{M}_{i+1}$. But that implies $\mathbb{M}_i \precnsim \mathbb{M}_{i+1}$, a contradiction.

The same argument applies to $\precsim^i$ as well.

5.2 Minimal Models for Protocol Analysis

When model-finding is used for protocol analysis, specifically when reasoning about an authentication goal, minimality with respect to arbitrary homomorphisms is of particular interest. Consider, for example, the analysis of the authentication properties of DoorSEP. The model $\mathbb{A}$ corresponding to the failure of authentication described in the Introduction is one in which there are keys for two *different* doors D and D' involved in the protocol run. The model $\mathbb{B}$ that would arise from identifying D and D' would still represent a protocol execution (indeed, the hoped-for behavior of the protocol). But $\mathbb{A}$ is strictly below this $\mathbb{B}$ in the $\precsim$ ordering, and it is $\mathbb{A}$ that gives insight in to the possibility of the man-in-the-middle attack (in the absence of the trust axiom, of course).

5.3 Computing Minimal Models and Set-of-Support

We present the following algorithms, each of which relies on the primitive operation of asking an SMT solver for a single finite model of a given theory. Recall that an SMTLib-compliant solver need not return any *particular* model for a

satisfiable theory, and that repeated requests to a solver for the same theory will typically return the same model.

Fix a theory T.

- iMinimize: given model $\mathbb{A} \models T$, compute an i-minimal model $\mathbb{M} \models T$ with $\mathbb{M} \precsim^i \mathbb{A}$.
- aMinimize: given model $\mathbb{A} \models T$, compute an a-minimal model $\mathbb{M} \models T$ with $\mathbb{M} \precsim \mathbb{A}$.
- computeCore: given model $\mathbb{A}$, compute the core of $\mathbb{A}$.
- SetOfSupport (resp. iSetOfSupport): compute a stream of models comprising a (resp. injective) set of support for theory T.
- aHomTo (resp. iHomTo): given model $\mathbb{A} \models T$, compute a sentence $homTo_{\mathbb{A}}$ defining the models $\mathbb{P} \models T$ such that there is a (resp. injective) homomorphism $h : \mathbb{P} \to \mathbb{A}$.
- aHomFrom (resp. iHomFrom): given model $\mathbb{A} \models T$, compute a sentence $homFrom_{\mathbb{A}}$ defining the models $\mathbb{P} \models T$ such that there is a (resp. injective) homomorphism $h : \mathbb{A} \to \mathbb{P}$.

The algorithms aMinimize and computeCore each rely on the sentences $homTo_{\mathbb{A}}$ and $homFrom_{\mathbb{A}}$. The latter of these is subtle, so **we first present the other algorithms in terms of these, then develop aHomTo and aHomFrom.**

5.4 *i*-Minimization

The following procedure was originally developed for use in the *Aluminum* tool [33] For this algorithm we use the notation $flip_{\mathbb{P}}$ to denote

$$\bigwedge\{\neg\alpha \mid \alpha \text{ is atomic}, \mathbb{P} \models \neg\alpha\} \wedge \bigvee\{\neg\beta \mid \beta \text{ is atomic}, \mathbb{P} \models \beta\}$$

Note in particular that if c and c' are constants naming distinct elements of a model $\mathbb{P}$, then $c \neq c'$ is one of the conjuncts of $flip_{\mathbb{P}}$.

Algorithm 12 (i-Minimize).

input: theory T and model $\mathbb{A} \models T$
output: model $\mathbb{P} \models T$ such that $\mathbb{P}$ is i-minimal for T and $\mathbb{P} \precsim^i \mathbb{A}$
initialize: set $\mathbb{P}$ to be $\mathbb{A}$
while $T' \stackrel{def}{=} T \cup \{flip_{\mathbb{P}}\}$ is satisfiable
  set $\mathbb{P}$ to be a model of T'
return $\mathbb{P}$

Lemma 13. *Algorithm 12 is correct: if $\mathbb{A}$ is a finite model of T then Algorithm 12 terminates on $\mathbb{A}$, and the output $\mathbb{P}$ is an i-minimal model of T with $\mathbb{P} \precsim^i \mathbb{A}$*

Proof. Each iteration goes down in the $\precnsim^i$ ordering, thus termination. To show that the result is i-minimal for T, it suffices to argue that the result is a minimal T-submodel of the input, under the submodel ordering. But this is clear from the definition of the sentences *flip*.

5.5 a-Minimization

Computing a-minimal models is harder. If we bound the size of the domain(s) of our models then a-minimal models exist: the $\precsim$ preorder is well-founded, so the set of minimal elements with respect to this order is non-empty. The question is, how to compute a-minimal models?

The idea is that, given a model $\mathbb{A}$, we can use the sentences $homTo_{\mathbb{A}}$ and $homFrom_{\mathbb{A}}$ to iterate the process of constructing a model that is strictly below $\mathbb{A}$ in the $\precsim$ ordering.

Algorithm 14 (a-Minimize).

input: theory T and model $\mathbb{A} \models T$
output: model $\mathbb{P} \models T$ such that $\mathbb{P}$ is a-minimal for T and $\mathbb{N} \precsim \mathbb{A}$
initialize: set $\mathbb{P}$ to be $\mathbb{A}$
while $T' \stackrel{def}{=} T \cup \{homTo_{\mathbb{P}}\} \cup \{\neg homFrom_{\mathbb{P}}\}$ is satisfiable
  set $\mathbb{P}$ to be a model of T'
return $\mathbb{P}$

Lemma 15. *Algorithm 14 is correct: if $\mathbb{A}$ is a finite model of T then Algorithm 12 terminates on $\mathbb{A}$, and the output $\mathbb{P}$ is an a-minimal model of T with $\mathbb{P} \precsim \mathbb{A}$*

Proof. Each iteration constructs a model lower in the $\precsim$ ordering; termination follows from well-foundedness of the $\precsim$ ordering.

5.6 Computing Cores

Cores are interesting for us because—when the input theory T is geometric—they give a way to build models that are both a-minimal and i-minimal.

Testing whether a model is a core is NP-complete [25]. So computing cores is presumably expensive, from a worst-case complexity perspective. But it is not difficult, using an SMT solver, to write a program that behaves well in practice. The key point is the well-known observation that a model $\mathbb{C}$ has no proper retracts if and only if it has no proper endomorphisms.

Definition 16. *If $\mathbb{A}$ is a finite model for signature Σ, the sentence $endo_{\mathbb{A}}$, over the signature Σ_h that extends Σ by adding a new function symbol $h_s : S \to S$ at each sort S, is the conjunction of*

- *the diagram of $\mathbb{A}$,*
- *the sentence expressing "h is a homomorphism", and*
- *the sentence expressing "h is not injective."*

Algorithm 17 (ComputeCore).

input: model $\mathbb{A}$ over signature Σ
output: a core $\mathbb{P}$ of $\mathbb{A}$
initialize: Set $\mathbb{P}$ to be $\mathbb{A}$

while $endo_{\mathbb{P}}$ is satisfiable
let $\mathbb{P}'$ be a model of $endo_{\mathbb{P}}$;
let $\mathbb{P}_0$ be the image of $endo_{\mathbb{P}}$ in $\mathbb{P}$';
let $\mathbb{P}$ be the reduct of $\mathbb{P}_0$ to the original signature Σ
return $\mathbb{P}$

Lemma 18. *Algorithm 17 computes a core of its input.*

Proof. The algorithm terminates because the size of the model $\mathbb{P}$ decreases at each iteration. The resulting model is a core, since it has no proper endomorphisms.

5.7 Set of Support

We take the ability to generate a set-of-support for the class of all models of a theory T to be a natural notion of "completeness" in model-finding. Lemma 8 makes a precise claim of completeness with respect to reasoning about geometric consequences of T.

It should be noted that if a class $\mathcal{C}$ is a set-of-support for a theory T with respect to i-homomorphisms then $\mathcal{C}$ is a set-of-support for T with respect to a-homomorphisms; this is immediate from the definitions.

There will be typically many more models comprising an i-set of support. However, it is true that if there is a *finite* $\mathcal{C}$ that is a set-of-support for a theory T with respect to a-homomorphisms then there is a finite $\mathcal{C}'$ set-of-support for T with respect to i-homomorphisms. To see this, suppose $\mathcal{C}$ is a set of support for a class of models. Each $\mathbb{A}$ in this set has a finite number of i-minimal models $B_1, \ldots B_k$ below it. The collection of all these taken over the models in $\mathcal{C}$ makes a i-set of support.

Computing sets-of-support is another application of the $homFrom_{\mathbb{A}}$ technique. Given theory T and model $\mathbb{A}$, if we construct the theory $T' \stackrel{def}{=} T \cup \{\neg homFrom_{\mathbb{A}}\}$ then calls to the SMT solver on theory T' are guaranteed to return models of T outside the hom-cone of $\mathbb{A}$ if any exist. So a set-of-support for T can be generated by iterating this process.

Completeness of this strategy does not require that the models $\mathbb{A}$ we work with are minimal. But if we do work with minimal models there will be fewer iterations. We give SetOfSupport here, for iSetOfSupport simply use i-minimal models and the $iHomFrom_{\mathbb{A}}$ sentence.

*Algorithm 19 (*SetOfSupport*).*

input: theory T and profile prf
output: a stream $\mathbb{M}_1, \mathbb{M}_2, \ldots$ of minimal models of T such that for any prf-model $\mathbb{P} \models T$, there is some i such that $\mathbb{M}_i \precsim \mathbb{P}$.
initialize: set theory T^* to be T
while T^* is satisfiable
let $\mathbb{M}$ be an a-minimal model of T^*
output $\mathbb{M}$
set T^* to be $T^* \cup \neg homFrom_{\mathbb{M}}$

5.8 Hom-To

This is straightforward "solver programming". Given model $\mathbb{A}$, we want to characterize those $\mathbb{P}$ such that there is a hom $h : \mathbb{P} \to \mathbb{A}$, by constructing a sentence $homTo_{\mathbb{A}}$ axiomatizing such models.

Algorithm 20 (HomTo).

input: model $\mathbb{A}$ over signature Σ.
output: sentence $homTo_{\mathbb{A}}$ in an expanded signature Σ^+, such that for any model $\mathbb{P} \models \Sigma$, $\mathbb{P} \precsim \mathbb{A}$ iff there is an expansion $\mathbb{P}^+$ of $\mathbb{P}$ to Σ^+ with $\mathbb{P}^+ \models homTo_{mM}$.

define Σ^+ to be the extension of Σ obtained by
- adding a set of fresh constants naming elements of the domain of $\mathbb{A}$
- adding a function symbol $h_S : S \to S$ at each sort S

return $homTo_{\mathbb{A}}$ as the conjunction of the following sentences, one for each function symbol f and predicate R in Σ. Here $\vec{e}$ and e' range over the names for elements of $\mathbb{A}$.

$$\forall \vec{x}, y.\ f\vec{x} = y \implies \bigvee \{(\vec{hx} = \vec{e} \wedge y = e') \mid \mathbb{A} \models f\vec{e} = e'\}$$

$$\forall \vec{x}.\ R\vec{x} = true \implies \bigvee \{(\vec{hx} = \vec{e}) \mid \mathbb{A} \models R\vec{e} = true\}$$

For $iHomTo_{,}$ simply add a sentence to say that h is injective.

Lemma 21. *There is a homomorphism from $\mathbb{B}$ to $\mathbb{A}$ iff there is a model $\mathbb{B}^+ \models homTo_{\mathbb{A}}$ such that $\mathbb{B}$ is the reduction to Σ of $\mathbb{B}^+$.*

5.9 Hom-From

Our eventual goal is: given a model $\mathbb{A}$, find a formula to capture **not** being in the hom-cone of $\mathbb{A}$. This is more interesting than the aHomTo problem, because we are going to *negate* the sentence we build, to express hom-cone-avoidance. Since universal quantifiers can be bottlenecks in SMT-solving, we want to minimize the number of existential quantifiers we use here.

The ideal outcome would be to construct an existential sentence capturing the complement of the hom cone of $\mathbb{A}$. Equivalently we might look for a structure $\mathbb{D}$ such that for any $\mathbb{X}$, $\mathbb{X} \precsim \mathbb{D}$ iff $\mathbb{A} \not\precsim \mathbb{X}$. This is called "homomorphism duality" in the literature [15]. Such a structure doesn't always exist, and even if it does, it can be exponentially large in the size of $\mathbb{A}$ [15]. So we turn to heuristic methods.

Our strategy is to construct a sentence (to be negated) which is guaranteed to characterize models in the hom-cone of $\mathbb{M}$, then refine this sentence to eliminate (some) quantifiers. We start with the equations of the standard model representation for $\mathbb{A}$ as described in Sect. 2. By replacing the Razor-defined constants by existentially-quantified variables we arrive at a sentence $rep_{\mathbb{A}}$, which is a positive-existential sentence (without disjunctions).

By the fact that homomorphisms preserve positive existential formulas and the fact that the equations of $rep_{\mathbb{A}}$ completely describe the functions and predicates true of $\mathbb{A}$ we have:

Lemma 22. *Let* $\mathbb{A}$ *and* $\mathbb{F}$ *be* Σ *models. Then* $\mathbb{A} \precsim \mathbb{F}$ *iff* $\mathbb{F} \models rep_{\mathbb{A}}$.

The trouble with $rep_{\mathbb{A}}$ is that it has as many existential quantifiers in $rep_{\mathbb{A}}$ as there are domain elements. If we were to take $homFrom_{\mathbb{A}}$ to be $rep_{\mathbb{A}}$, simply negating this would lead to a sentence inconvenient for the SMT solver. We can compress the representation, though. This will lead to a nicer representation sentence, which we will take as our $homFrom_{\mathbb{M}}$.

Algorithm 23 (HomFrom).

input: model $\mathbb{A}$ over signature Σ
output: sentence $homFrom_{\mathbb{A}}$ over signature Σ, such that for any model $\mathbb{P} \models \Sigma$, $\mathbb{A} \precsim \mathbb{P}$ iff $\mathbb{P} \models homFrom_{\mathbb{A}}$.
comment: sentence $homFrom_{\mathbb{A}}$ is designed to use as few existential quantifiers as possible, in a "best-effort" sense.
initialize: Set sentence $homFrom_{\mathbb{A}}$ to be $rep_{\mathbb{A}}$, the standard model representation sentence for $\mathbb{A}$.
while there is a conjunct in the body of $homFrom_{\mathbb{A}}$ of the form $f(t_1, \ldots, t_n) = x$ such that x does not occur in any of the t_i,

- replace all occurrences of x in $homFrom_{\mathbb{A}}$ by $f(t_1, \ldots, t_n)$. Erase the resulting trivial equation $f(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ and erase the $(\exists x)$ quantifier in front.

For *iHomFrom*, first enrich $rep_{\mathbb{A}}$ to say that each of the fresh constants naming elements of $\mathbb{A}$ is distinct. The rest of the development goes through as described.

Lemma 24. *For any model* $\mathbb{P} \models \Sigma$*,* $\mathbb{A} \precsim \mathbb{P}$ *iff* $\mathbb{P} \models homFrom_{\mathbb{A}}$*. Similarly for* $iHomFrom_{\mathbb{A}}$ *and* $\precsim^i$.

The order in which we do these rules matters, in the sense that smaller formulas result if we process nodes as follows. Construct a graph in which the nodes are the variables occurring in the set of equations, and in which, if $fx_1 \ldots x_n = x$ is a rule, then there is an edge from each x_i to x. Then process the nodes according to the preorder given by this graph.

Example 25. Start with

$$\sigma \equiv \exists x_0 x_1 x_2 \;.\; fx_0 = x_2 \wedge fx_1 = x_0 \wedge fx_2 = x_1 \wedge c = x_2$$

Making the graph as defined above, we treat the variables in the order x_2, then x_1 then x_0. We then derive, in order:

$$\exists x_0 x_1 \;.\; fx_0 = c \wedge fx_1 = x_0 \wedge fc = x_1$$
$$\exists x_0 \;.\; fx_0 = c \wedge ffc = x_0$$
$$fffc = c$$

An SMT solver will work more happily with $fffc \neq c$ than with $\neg\sigma$.

5.10 Section Summary

1. i-minimal models for a theory T always exist; there may be no finite a-minimal models for a given theory.
2. a-minimal models are better suited to protocol analysis since they do not make unnecessary identifications between terms.
3. i-minimal models are easier to compute than a-minimal models.
4. If T is a geometric theory, and $\mathbb{M}$ is an a-minimal model and a core, then $\mathbb{M}$ is i-minimal (Lemma 10).
5. If a class $\mathcal{C}$ is a set-of-support for a theory T with respect to i-homomorphisms then $\mathcal{C}$ is a set-of-support for T with respect to a-homomorphisms.
6. If there is a finite $\mathcal{C}$ that is a set-of-support for a theory T with respect to a-homomorphisms then there is a finite $\mathcal{C}'$ set-of-support for T with respect to i-homomorphisms.

6 Conclusions and Future Work

In this paper, we have developed a method for analyzing systems with cryptographic protocols in the context of first-order theories such as trust assumptions, and presented a detailed analysis of a specific example, the DoorSEP protocol.

We have described an implementation of these methods as the Logical Protocol Analysis (LPA) system. LPA is a coordination between a general-purpose model-finder, Razor, and a cryptographic protocol-specific tool, CPSA. We have shown how to share labor between Razor and CPSA so that the latter can apply its authentication test solving methods, while Razor is handling the remainder of the axiomatic theory of the protocol together with some non-protocol axioms.

We explored the comparative virtues of minimality with respect to injective homomorphisms versus arbitrary homomorphisms, and developed algorithms for finding minimal models and computing a set-of-support of models for a theory.

Unfortunately, as the size of a protocol grows, so does the size of its theory, and SMT solvers struggle with performance in the presence of a significant number of universal quantifiers. In future work, we plan to reorganize the software architecture to be one in which only subtheories are delivered to the solver, preferably governing smaller parts of the domain.

References

1. Abramsky, S.: Domain theory in logical form. Ann. Pure Appl. Logic **51**, 1–77 (1991)
2. Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-LIB standard: version 2.0. In: Proceedings of 8th International Workshop on Satisfiability Modulo Theories, vol. 13, p.14 (2010)
3. Baumgartner, P., Fuchs, A., De Nivelle, H., Tinelli, C.: Computing finite models by reduction to function-free clause logic. J. Appl. Logic **7**, 58–74 (2009)
4. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Pironti, A., Strub, P.-Y.: Triple handshakes and cookie cutters: breaking and fixing authentication over TLS. In: IEEE Symposium on Security and Privacy (2014)

5. Blanchet, B.: From secrecy to authenticity in security protocols. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 342–359. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45789-5_25
6. Blanchet, B.: Automatic proof of strong secrecy for security protocols. In: Proceedings of the 2004 IEEE Symposium on Security and Privacy, pp. 86–100. IEEE CS Press, May 2004
7. Blanchet, B.: Vérification automatique de protocoles cryptographiques: modèle formel et modèle calculatoire. Habilitation thesis, Université Paris-Dauphine, November 2008
8. Blanchet, B., Abadi, M., Fournet, C.: Automated verification of selected equivalences for security protocols. J. Log. Algebr. Program. **75**(1), 3–51 (2008)
9. Chadha, R., Cheval, V., Ciobâcă, Ş., Kremer, S.: Automated verification of equivalence properties of cryptographic protocols. ACM Trans. Comput. Log. **17**(4), 23:1–23:32 (2016)
10. Claessen, K., Sörensson, N.: New techniques that improve MACE-style finite model finding. In: CADE Workshop on Model Computation-Principles, Algorithms, Applications (2003)
11. Cremers, C., Mauw, S.: Operational Semantics and Verification of Security Protocols. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-540-78636-8
12. 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). https://doi.org/10.1007/978-3-540-78800-3_24
13. Dolev, D., Yao, A.: On the security of public-key protocols. IEEE Trans. Inf. Theory **29**, 198–208 (1983)
14. Dougherty, D.J., Guttman, J.D., Ramsdell, J. D.: Homomorphisms and minimality for enrich-by-need security analysis. arXiv.org http://arxiv.org/abs/1804.07158, April 2018
15. Erdös, P.L., Pálvölgyi, D., Tardif, C., Tardos, G.: Regular families of forests, antichains and duality pairs of relational structures. Combinatorica **37**(4), 651–672 (2017)
16. Escobar, S., Meadows, C., Meseguer, J.: Maude-NPA: cryptographic protocol analysis modulo equational properties. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2007-2009. LNCS, vol. 5705, pp. 1–50. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03829-7_1
17. Fagin, R., Kolaitis, P., Popa, L.: Data exchange: getting to the core. ACM Trans. Database Syst. (TODS) **30**(1), 174–210 (2005)
18. Fournet, C., Gordon, A.D., Maffeis, S.: A type discipline for authorization policies. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 141–156. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_11
19. Gordon, A.D., Pucella, R.: Validating a web service security abstraction by typing. Formal Asp. Comput. **17**(3), 277–318 (2005)
20. Guttman, J.D.: Shapes: surveying crypto protocol runs. In: Cortier, V., Kremer, S. (eds.) Formal Models and Techniques for Analyzing Security Protocols, Cryptology and Information Security Series. IOS Press (2011)
21. Guttman, J.D.: Establishing and preserving protocol security goals. J. Comput. Secur. **22**(2), 201–267 (2014)

22. Guttman, J.D., Herzog, J.C., Ramsdell, J.D., Sniffen, B.T.: Programming cryptographic protocols. In: De Nicola, R., Sangiorgi, D. (eds.) TGC 2005. LNCS, vol. 3705, pp. 116–145. Springer, Heidelberg (2005). https://doi.org/10.1007/11580850_8
23. Guttman, J.D., Thayer, F.J.: Authentication tests and the structure of bundles. Theoret. Comput. Sci. **283**(2), 333–380 (2002)
24. Guttman, J.D., Thayer, F.J., Carlson, J.A., Herzog, J.C., Ramsdell, J.D., Sniffen, B.T.: Trust management in strand spaces: a Rely-Guarantee method. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 325–339. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24725-8_23
25. Hell, P., Nešetřil, J.: The core of a graph. Discret. Math. **109**(1–3), 117–126 (1992)
26. Jackson, D.: Software Abstractions, 2nd edn. MIT Press, Cambridge (2012)
27. Liskov, M.D., Rowe, P.D., Thayer, F.J.: Completeness of CPSA. Technical Report MTR110479, The MITRE Corporation, March 2011. http://www.mitre.org/publications/technical-papers/completeness-of-cpsa
28. Lobo, J., Minker, J., Rajasekar, A.: Foundations of Disjunctive Logic Programming. MIT Press, Cambridge (1992)
29. McCune, W.: MACE 2.0 Reference Manual and Guide. CoRR (2001)
30. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN prover for the symbolic analysis of security protocols. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 696–701. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_48
31. Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: USENIX Large Installation System Administration Conference (2010)
32. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: principled scenario exploration through minimality. In: International Conference on Software Engineering (2013)
33. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: principled scenario exploration through minimality. In: 35th International Conference on Software Engineering (ICSE), pp. 232–241 (2013)
34. Ramsdell, J.D.: Deducing security goals from shape analysis sentences. The MITRE Corporation, April 2012. http://arxiv.org/abs/1204.0480
35. Ramsdell, J.D., Guttman, J.D.: CPSA4: a cryptographic protocol shapes analyzer (2017). https://github.com/ramsdell/cpsa
36. Ramsdell, J.D., Guttman, J.D., Liskov, M.: CPSA: a cryptographic protocol shapes analyzer (2016). http://hackage.haskell.org/package/cpsa
37. Rescorla, E., Ray, M., Dispensa, S., Oskov, N.: Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), February 2010
38. Reynolds, A., Tinelli, C., Goel, A., Krstić, S.: Finite model finding in SMT. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 640–655. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_42
39. Robinson, A.: Handbook of Automated Reasoning, vol. 2. Elsevier, Hoboken (2001)
40. Rossman, B.: Homomorphism preservation theorems. J. ACM (JACM) **55**(3), 15 (2008)

41. Rowe, P.D., Guttman, J.D., Liskov, M.D.: Measuring protocol strength with security goals. Int. J. Inf. Secur. (2016). https://doi.org/10.1007/s10207-016-0319-z. http://web.cs.wpi.edu/~guttman/pubs/ijis_measuring-security.pdf
42. Saghafi, S., Danas, R., Dougherty, D.J.: Exploring theories with a model-finding assistant. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 434–449. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_30
43. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_49

A Precise Pictorial Language for Array Invariants

Johannes Eriksson, Masoumeh Parsa(✉), and Ralph-Johan Back

Department of Information Technologies, Åbo Akademi University,
Vesilinnantie 3, 20500 Turku, Finland
Johannes.Eriksson@fourferries.fi, {mparsa,backrj}@abo.fi

Abstract. Pictorial languages, while intuitive and descriptive, are rarely used as the primary reasoning language in program verification due to lack of precision. In this paper, we introduce a precise pictorial language for specifying array invariants that preserves visual perspicuity. The language extends Reynold's *partition diagrams* with the notion of a *coloring*, allowing assertions over portions of an array to be expressed by color-coding. The semantics of a coloring is given by a *legend*, mapping a colored partition of an array into a universally quantified predicate over the array. The pictorial syntax is an extension to*invariant diagrams*, transition graphs where preconditions, postconditions and invariants, rather than the program code, determine the main program structure. We demonstrate the approach with three examples, verified using the Why3 theorem prover frontend.

1 Introduction

Deductive program verification is the process of establishing correctness by proving verification conditions (VCs) extracted from a program. It relies on a formal pre- and postcondition specification as well as loop invariants being provided by the programmer. This task by itself requires proficiency in mathematical logic. Further challenges include assessing completeness of the specification, whether invariants are sufficiently strong to establish the postcondition while sufficiently weak to be maintained, and using automatic theorem provers. Training in formal methods aims at giving the necessary conceptual and technical skills to address these challenges.

In instructional settings, verification is often taught by examples from tangible and visually perspicuous domains, such as arrays of colored objects in the case of the famous Dutch national flag three-way-partitioning problem [1]. While such examples are a valuable pedagogical device, how to generalize the reasoning to more typical programming problems is often left unexplained. Also, the transition from instructional pictures to a mathematically precise *reasoning language* does not always follow a happy path. Even though influential authors have already several decades ago highlighted the benefits of pictures in formal reasoning [2,3], pictures have by and large been employed as stepping stones towards

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 151–160, 2018.
https://doi.org/10.1007/978-3-319-98938-9_9

some final, textual, formalization suitable for conventional symbolic reasoning. While reasons therefore (lack of precision, technical limitations, convention) may be legitimate, we suspect that this demoted role of pictures means their full benefit as reasoning tools is not being realized.

A notable exception is Reynold's *interval* and *partition diagrams* [4], which integrate pictures with mathematical notation seamlessly, allowing invariants and even proofs over arrays to be expressed in a way that simultaneously maintains visual perspicuity and mathematical precision. It is on this trajectory that we position the approach described in this paper. A partition diagram, in its base form, is a *precise*, *compact* and *embeddable* diagram stating that a collection of integer indexes subdivides an array into disjoint partitions. Precise means that the language has a well-defined mathematical meaning, compact that it is space-conserving, and embeddable that it can be integrated into another diagram or a textual formula. Associated with a partitioning is some collection of properties, that the elements in the partitions should satisfy. A property can be expressed precisely by a formula universally quantifying over the partition diagram (e.g., [4, p. 94]), or by (less formally) annotating the partition diagrams with the properties (e.g., [3, p. 94]). In line with the second approach, we extend partition diagrams with the notion of *coloring* a partition. Formally, a coloring is a function from array indexes to a small finite set ("palette") of programmer-defined colors. The programmer gives interpretation to the colors through the *legend* construct. Analogously to its cartography namesake, a legend is a mapping from colors to a universally-quantified predicate over the colored partitions. Together, partitionings, colorings and legends provide a precise and expressive pictorial language for array invariants.

As an umbrella framework we use *invariant-based programming* (IBP), a correct-by-construction formal verification approach geared towards teaching [5]. In IBP, preconditions, postconditions and invariants—under the common nomen *situations*—serve as the main organizing structures of a program. The program is represented by an *invariant diagram*, a graph of nested situations connected by *transitions*. The situations represent state predicates, such as pre- and postconditions and invariants, while the transitions constitute the actual executable code. We define the semantics of colorings and legends by translation into predicates over the program state. After translation, the VCs of the diagram are extracted using the proof rules of invariant diagrams. Nesting allows substitutions to inherit constraints from outer situations. In our extension, nesting also allows legends to be shared by multiple situations, as well as to be extended in substitutions with additional color interpretations. We illustrate the approach with examples from the domain of searching and sorting. The examples have been mechanically verified using the Why3 platform [6], a front-end for a number of automatic theorem provers.

We proceed as follows. Section 2 introduces the pictorial language in the context of two search programs. Section 3 describes the verification semantics. A verification of a slightly more complex program is given in Sect. 4. We discuss related work in Sect. 5 and end the paper with conclusions and future work in Sect. 6.

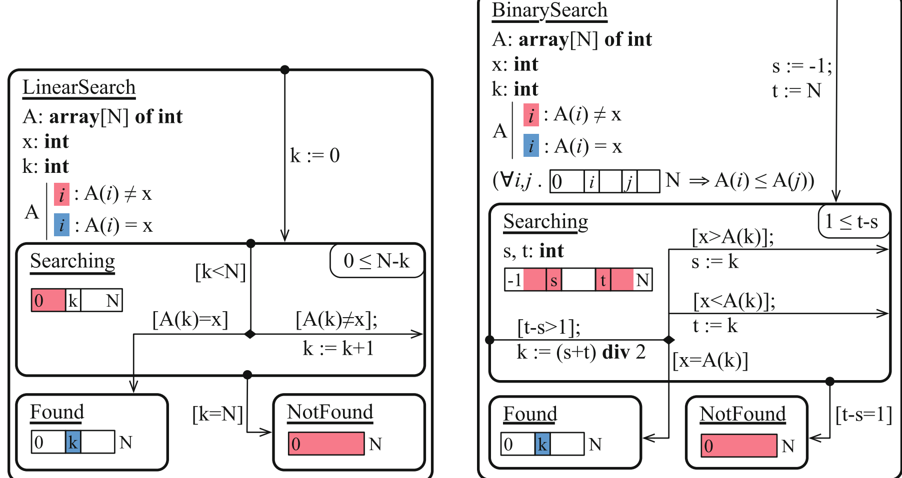


Fig. 1. Linear and binary search

2 Pictorial Invariant Diagrams

Consider the leftmost invariant diagram in Fig. 1. Each rounded rectangle—called a *situation*—identifies a subset of all possible program states. The role of a situation in a program is determined by the *transitions*, guarded program statements, connecting to it: a situation with no incoming transitions corresponds to a precondition (LinearSearch); a situation with no outgoing transitions corresponds to a postcondition (Found and NotFound). A situation with both incoming and outgoing transitions is an *intermediate situation*; an intermediate situation (or collection of intermediate situations) connected through a cycle of transitions corresponds to a *loop* (Searching). There are five types of declarations that can appear inside a situation:

Variable declarations introduce program variables and associate them with types. For example, the declaration "A :**array**[N] **of int**" in the situation LinearSearch types the variable A as an integer array of length N, indexed from 0 to $N - 1$.

Legends introduce colors and assign them their meanings. For instance, the legend "A| *i* : A(i) $\neq$ x" states that the red elements in A are different from x. A legend is not a state assertion; rather it introduces an implication, allowing invariants over an array to be expressed visually by "painting" sub-arrays with a relevant property (in this case, that the sub-array is known to not hold the value x). We make this notion more precise in the next section. Legends may introduce any number of new colors, but the color palettes for distinct arrays must be disjoint.

Invariants are assertions over the program variables of the situation. We can express invariants using standard mathematical and logical notation. E.g., "$0 \leq \mathrm{k} \leq \mathrm{N}$" expresses that the value of k is between 0 and N, inclusively. For asserting that a collection of variables form a partitioning we prefer to use Reynolds-style partition diagrams. The basic partition diagram is a rectangular contour:

$$i\boxed{\quad\; j}$$

where i and j are integer expressions over the program variables. It stands for the predicate "$i < j$". The bounds may be juxtaposed with respect to the adjacent edge to specify whether they are inclusive or exclusive:

$$
\begin{aligned}
i\boxed{\quad\;}j &= i\boxed{\quad j-1} &&= i < j-1\\
\boxed{i\quad\;}j &= i-1\boxed{\quad j-1} &&= i-1 < j-1\\
\boxed{i\quad j} &= i-1\boxed{\quad j} &&= i-1 < j
\end{aligned}
$$

Conjunctions of partition diagrams, when the upper bound of the predecessor coincides with the lower bound of its successor, may be written in chained form:

$$
\begin{aligned}
i\boxed{\quad j}\boxed{\quad k} &= i\boxed{\quad j} \wedge j\boxed{\quad k}\\
i\boxed{\quad\;}\boxed{j}\boxed{\quad k} &= i\boxed{\quad j} \wedge \boxed{j\quad k}
\end{aligned}
$$

The following abbreviations denote singleton intervals:

$$
\begin{aligned}
\boxed{i} &= \boxed{i\quad i}\\
i\boxed{\,} &= \boxed{i+1\quad i+1}\\
\boxed{\,}i &= \boxed{i-1\quad i-1}
\end{aligned}
$$

Using partition diagrams, the aforementioned predicate is equivalently expressed as:

$$\boxed{0\quad\;}\boxed{\mathrm{k}}\boxed{\quad\; \mathrm{N}} = 0 \leq \mathrm{k} \leq \mathrm{N}$$

As mentioned, partition diagrams can be embedded in textual formulas; e.g., the invariant of BinarySearch states that A is sorted.

Colorings are pictorial invariants similar to partition diagrams, but appear as colored regions rather than as contours. The basic form is

$$i\;\colorbox{lightgray}{$\quad /c/ \quad j$}$$

where c is the chosen color of the regions (for contrast, we chiefly pick ■, ■ and ■). It stands for the partial definition of a *coloring function* over the integer interval $(i, j]$. Colorings allow the same syntactic shorthands as partition diagrams (bound juxtaposition, chaining and singleton intervals). For example, the following coloring asserts that the coloring function takes the value ■ between 0 (inclusive) and k (exclusive):

$$\colorbox{pink}{$0\quad\;$}\,\mathrm{k}$$

For compact representation, partitioning and coloring invariants may be drawn overlapping when their bounds coincide. For example, the invariant of situation BinarySearch ▷ Searching is the conjunction of a partitioning and a coloring:

0	k	N

=

0	k	N

∧ 0 k

Variants are written in the upper right corner of intermediate situations that are part of a loop. To verify termination, we need to show that the variant $(\mathrm{N} - k)$ is decreased by each transition through the situation and does not decrease below the lower bound (0).

Finally, we note that situations can be nested. Nesting is *conjunctive*: an inner situation inherits all declarations, with the exception of variants, from the enclosing situations.

3 Verification of Pictorial Invariant Diagrams

An invariant diagram is *correct* iff it is *consistent*, *terminating* and *live*. A transition t from a situation satisfying predicate p to situation satisfying predicate q is consistent if $p \Rightarrow \mathsf{wp}(t, q)$ is true, where wp is the weakest precondition transformer. For termination, we check that the variant v decreases and that its lower bound is maintained on re-entry to the situation, i.e., $v = v_0 \wedge p \Rightarrow \mathsf{wp}(t, 0 \leq v < v_0)$. A situation satisfying p is live if at least one outgoing transition is always enabled, i.e, $p \Rightarrow \mathsf{wp}(t, g_1 \vee \cdots \vee g_n)$, where $g_1, \ldots, g_n$ are the guards of outgoing transitions. Next, we describe how the pictorial elements of a situation (legends and colorings) combine into a predicate onto which these rules can be applied. For a formal treatment of the proof rules themselves, see [7].

For a given situation s, let $\overline{x}$ be the declared variables, $\overline{T}$ their types, and $\overline{a}$ the subset of $\overline{x}$ containing only the variables of array type. The *coloring function* associated with a variable $A \in \overline{a}$ of type **array**[N] in situation s is a total function from the program state and an array index

$$\mathsf{col}_{s,A} : \overline{T} \times [0, \mathrm{N}) \rightarrow C_{s,A} \cup \{\boxslash\}$$

where the set $C_{s,A}$ is the color palette associated with A in s, and $\boxslash$ is a special value indicating that no coloring has been specified. The coloring function formalizes the mapping between legends and invariants, is fully defined, and is intended to be fully eliminated from the final VC. Given the colorings declared for array A in situation s:

$$i_1 \; /c_1/ \; j_1 \; \ldots \; i_n \; /c_n/ \; j_n$$

where $c_1, \ldots, c_n \in C_{s,A}$, the coloring function is defined as:

$$\begin{aligned} \mathsf{col}_{s,A}(\overline{x})(i) = \; & \mathsf{if}\ (i_1 < i \leq j_1)\ \mathsf{then}\ c_1 \\ & \vdots \\ & \mathsf{else\ if}\ (i_n < i \leq j_n)\ \mathsf{then}\ c_n \\ & \mathsf{else}\ \boxslash \end{aligned}$$

Disjointness of partitioning means that the if-conditions are mutually exclusive, and the else-clause ensures that the function is total. A legend declaration for variable a in situation s has the general form:

$$A \mid i_1 \;\boxed{\;/c_1/\;}\; j_1 \;\ldots\; i_n \;\boxed{\;/c_n/\;}\; j_n \;:\; p(\overline{x}, i_1, \ldots, i_n, j_1, \ldots, j_n)$$

where $c_1, \ldots, c_n \in C_{s,a}$ and p is a predicate on the program state. Semantically, this legend stands for the following predicate:

$$\begin{aligned}\mathsf{lgd}_{s,A}(\overline{x}) = (\forall i_1, \ldots, i_n, j_1, \ldots, j_n \;\; . \;\; & (0 \leq i_1 < j_1 < \mathrm{N}) \wedge \cdots \wedge (0 \leq i_n < j_n < \mathrm{N}) \\ \wedge \;\; & (\forall k \,.\, i_1 < k \leq j_1 \Rightarrow \mathsf{col}_{s,A}(\overline{x})(k) = c_1) \\ & \vdots \\ \wedge \;\; & (\forall k \,.\, i_n < k \leq j_n \Rightarrow \mathsf{col}_{s,A}(\overline{x})(k) = c_n) \\ \Rightarrow \;\; & p(\overline{x}, i_1, \ldots, i_n, j_1, \ldots, j_n))\end{aligned}$$

That is, a legend is an assertion that p holds for subintervals of A matching the sequence of colorings given in the legend. Like invariants, legends are conjunctive.

To verify a diagram, we generate a theory including the coloring functions, legend predicates and invariants of each situation, and a lemma to be proved for each transition. For example, the theory of $\underline{\text{BinarySearch}} \rhd \underline{\text{Searching}}$ in Fig. 1 contains the following declarations (for brevity, in $\mathsf{lgd}_{\underline{\text{Searching}},A}$ we have omitted the inner quantifications, as the ranges are singletons in both cases):

$$\begin{aligned}\mathsf{col}_{\underline{\text{Searching}},A}(A,s,t)(i) &= \mathsf{if}\ (-1 \leq i \leq s)\ \mathsf{then}\ {\color{red}\blacksquare} \\ &\ \mathsf{else\ if}\ (t \leq i < N)\ \mathsf{then}\ {\color{red}\blacksquare} \\ &\ \mathsf{else}\ \boxslash \\ \mathsf{lgd}_{\underline{\text{Searching}},A}(A,s,t) &= \quad (\forall i \,.\, (0 \leq i < N) \wedge (\mathsf{col}_{\underline{\text{Searching}},A}(i) = {\color{red}\blacksquare}) \Rightarrow A(i) \neq x) \\ &\ \wedge (\forall i \,.\, (0 \leq i < N) \wedge (\mathsf{col}_{\underline{\text{Searching}},A}(i) = {\color{blue}\blacksquare}) \Rightarrow A(i) = x) \\ \mathsf{inv}_{\underline{\text{Searching}},A}(A,s,t) &= \quad -1 \leq s < t \leq N \\ &\ \wedge (\forall i,j . 0 \leq i \leq j \leq N \Rightarrow A(i) \leq A(j))\end{aligned}$$

To generate the VCs for situation $\underline{\text{Searching}}$, we can now apply the proof rules of IBP, taking the conjunction of $\mathsf{lgd}_{\underline{\text{Searching}},A}$ and $\mathsf{inv}_{\underline{\text{Searching}},A}$ as the situation predicate. For example, to prove that the loop transition "$[\mathrm{t} - \mathrm{s} > 1];\ \mathrm{k} := (\mathrm{s} + \mathrm{t})\ \mathsf{div}\ 2;\ [\mathrm{x} > \mathrm{A}(\mathrm{k})];\ \mathrm{s} := \mathrm{k}$" is consistent we will need to discharge the following VC:

$$\begin{aligned}(\forall A, s, t \;\; . \;\; & \mathsf{lgd}_{\underline{\text{Searching}},A}(A,s,t) \wedge \mathsf{inv}_{\underline{\text{Searching}},A}(A,s,t) \\ & \wedge\ t - s > 1\ \wedge\ k = (s+t)\ \mathsf{div}\ 2\ \wedge\ x > A(k)\ \wedge\ s' = k \\ & \Rightarrow \mathsf{lgd}_{\underline{\text{Searching}},A}(A,s',t) \wedge \mathsf{inv}_{\underline{\text{Searching}},A}(A,s',t))\end{aligned}$$

Additionally, to prove that the same transition is decreasing the variant of $\underline{\text{Searching}}$:

$$\begin{aligned}(\forall A, s, t \;\; . \;\; & \mathsf{lgd}_{\underline{\text{Searching}},A}(A,s,t) \wedge \mathsf{inv}_{\underline{\text{Searching}},A}(A,s,t) \\ & \wedge\ t - s > 1\ \wedge\ k = (s+t)\ \mathsf{div}\ 2\ \wedge\ x > A(k)\ \wedge\ s' = k \\ & \Rightarrow 1 \leq t - s' < t - s)\end{aligned}$$

Note that the antecedents are identical to those of the consistency VC. Finally, the liveness condition for situation Searching is:

$$(\forall A, s, t \ . \ \mathsf{lgd}_{\underline{\text{Searching}},A}(A, s, t) \wedge \mathsf{inv}_{\underline{\text{Searching}},A}(A, s, t) \\ \wedge \ k = (s + t) \ \mathsf{div}\ 2 \\ \Rightarrow (t - s = 1) \vee (t - s > 1 \wedge (x > A(k) \vee x < A(k) \vee x = A(k))))$$

The VCs can now be discharged using an automatic theorem prover.

4 Example: Insertion Sort

Figure 2 shows an invariant diagram interpretation of insertion sort. It consists of an outer loop (Sorting) maintaining a sorted partition (green), and an inner loop (Inserting) moving the next element from the unsorted partition into its correct position in the sorted partition. The inner loop, as it moves the element back one step per iteration, maintains two sorted partitions (green and blue). The control flow transfers from the inner to the outer loop when the concatenation of the partitions becomes sorted. The outer loop terminates when every element of the array has been processed. Transitions must additionally ensure that A is a permutation of the original A_0.

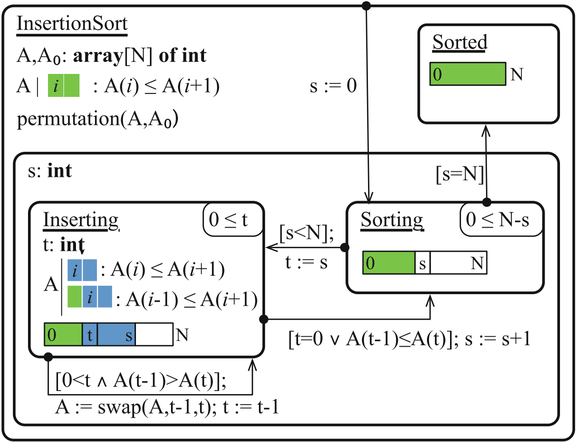


Fig. 2. Insertion sort (Color figure online)

Like invariants, legends are inherited from outer situations. For instance, that any two adjacent green elements are sorted is visible to both Sorting and Inserting. We note that legends may introduce new colors limited in scope to the declaring situation and its nested situations. For example, blue introduced by the legend of Inserting is visible only within Inserting. The coloring functions, legend

$$
\begin{aligned}
\mathsf{col}_{\underline{\text{Sorting}},A}(A,A_0,s)(i) &= \text{if } (0 \le i < s) \text{ then } {\color{green}\blacksquare} \text{ else } \boxslash \\
\mathsf{lgd}_{\underline{\text{Sorting}},A}(A,A_0,s) &= (\forall i \;.\; 0 \le i < N \wedge 0 \le i+1 < N \\
&\qquad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i) = {\color{green}\blacksquare} \\
&\qquad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i+1) = {\color{green}\blacksquare} \\
&\qquad \Rightarrow A(i) \le A(i+1)) \\
\mathsf{inv}_{\underline{\text{Sorting}},A}(A,s) &= \mathrm{permutation}(A,A_0) \wedge 0 \le s \le N \\
\mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i) &= \text{if } (0 \le i < t) \text{ then } {\color{green}\blacksquare} \\
&\quad\ \text{else if } (t \le i \le s) \text{ then } {\color{blue}\blacksquare} \\
&\quad\ \text{else } \boxslash \\
\mathsf{lgd}_{\underline{\text{Inserting}},A}(A,A_0,s,t) &= \quad (\forall i \;.\; 0 \le i < N \wedge 0 \le i+1 < N \\
&\qquad\quad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i) = {\color{green}\blacksquare} \\
&\qquad\quad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i+1) = {\color{green}\blacksquare} \\
&\qquad\quad \Rightarrow A(i) \le A(i+1)) \\
&\quad\ \wedge\ (\forall i \;.\; 0 \le i < N \wedge 0 \le i+1 < N \\
&\qquad\quad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i) = {\color{blue}\blacksquare} \\
&\qquad\quad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i+1) = {\color{blue}\blacksquare} \\
&\qquad\quad \Rightarrow A(i) \le A(i+1)) \\
&\quad\ \wedge\ (\forall i \;.\; 0 \le i-1 < N \wedge 0 \le i+1 < N \\
&\qquad\quad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i-1) = {\color{green}\blacksquare} \\
&\qquad\quad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i) = {\color{blue}\blacksquare} \\
&\qquad\quad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i+1) = {\color{blue}\blacksquare} \\
&\qquad\quad \Rightarrow A(i) \le A(i+1)) \\
\mathsf{inv}_{\underline{\text{Inserting}},A}(A,s,t) &= \quad \mathrm{permutation}(A,A_0) \\
&\quad\ \wedge\ 0 \le t \le s < N
\end{aligned}
$$

Fig. 3. Coloring function, invariant and legend predicate of situations Sorting and Inserting. (Color figure online)

predicates and invariant predicates for situations Sorting and Inserting are shown in Fig. 3. Given these functions and predicates, the VCs for the transitions are formulated as described in the previous section (omitted here for brevity). The VCs are automatically proved by Why3 and its associated SMT solvers Z3 [8] and CVC4 [9].

5 Related Work

Reynolds [4] introduced interval and partition diagrams to express constraints on arrays. Gries's seminal textbook [3] uses array pictures in several examples. Astrachan [2] suggests pictorial representations of arrays and linked lists. Ginat [10] considers loop invariants as mathematical games, with emphasis on the heuristics of invariant identification. Some recent approaches have explored transforming invariant problems into games [11,12] and crowdsourcing verification to online communities. Partitioning has been employed in static analysis and heuristics-driven loop invariant generation [13,14]. Reasoning on range predicates is the basis of the axiomatic rules on array manipulations for correctness proofs of programs involving arrays in [15]. The converse problem, generating

visual representations from textual specifications, has been addressed in the context of the Z language [16], and also with the purpose of visualizing VCs on arrays [17]. While pictures and colors are a staple in algorithm animation, we are not aware of prior work combining partitionings and colorings for formal reasoning.

6 Conclusions and Future Work

In this paper, we have introduced a pictorial language for invariants over arrays. The language extends two existing visual formalisms: the notation for invariants and predicates builds on Reynold's partition diagrams, extending them with colorings to connect partitions with desired properties; the language for specifying the invariant structure and program statements is invariant diagrams, extended with a hierarchical mapping of colorings to predicates. Partition diagrams, colorings and legends seem to be rather expressive visual constructs, allowing many common array invariants to be stated.

This work is in its initial phases with multiple directions to be explored. First and foremost, tool support (in the form of editors and VC generators) would be needed for practical use. Existing tools for IBP [18] do not support the array-specific visual notations introduced here. Secondly, we would like to generalize the approach to more advanced data structures, such as trees and graphs. One challenge here is finding equally expressive and intuitive visual partitioning notations to state invariants over these non-linear data structures. Thirdly, we believe that colorings could serve runtime visualization and animation by overlaying the colors on a data structure instance picture, and analogously, to produce color-coded counterexamples during verification.

References

1. Dijkstra, E.W.: A Discipline of Programming, 1st edn. Prentice Hall PTR, Upper Saddle River (1997)
2. Astrachan, O.L.: Pictures as invariants. In: Dale, N.B. (ed.) Proceedings of the 22nd SIGCSE Technical Symposium on Computer Science Education 1991, San Antonio, Texas, USA, 7–8 March 1991, pp. 112–118. ACM (1991)
3. Gries, D.: The Science of Programming, 1st edn. Springer, Secaucus (1987). https://doi.org/10.1007/978-1-4612-5983-1
4. Reynolds, J.C.: The Craft of Programming. Prentice Hall PTR, Upper Saddle River (1981)
5. Back, R.J.: Invariant based programming: basic approach and teaching experiences. Form. Asp. Comput. **21**(3), 227–244 (2009)
6. Filliâtre, J.C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
7. Back, R.J., Preoteasa, V.: Semantics and proof rules of invariant based programs. In: Proceedings of the 2011 ACM Symposium on Applied Computing, SAC 2011, pp. 1658–1665. ACM, New York (2011)

8. 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). https://doi.org/10.1007/978-3-540-78800-3_24
9. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
10. Ginat, D.: Loop invariants and mathematical games. SIGCSE Bull. **27**(1), 263–267 (1995)
11. Fava, D., Shapiro, D., Osborn, J., Schäef, M., Whitehead Jr., E.J.: Crowdsourcing program preconditions via a classification game. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, pp. 1086–1096. ACM, New York (2016)
12. Logas, H., Vallejos, R., Osborn, J., Compton, K., Whitehead, J.: Visualizing loops and data structures in Xylem: the code of plants. In: Proceedings of the Fourth International Workshop on Games and Software Engineering, GAS 2015, pp. 50–56. IEEE Press, Piscataway (2015)
13. Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, pp. 338–350. ACM, New York (2005)
14. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, pp. 105–118. ACM, New York (2011)
15. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_23
16. Moremedi, K., van der Poll, J.A.: Transforming formal specification constructs into diagrammatic notations. In: Cuzzocrea, A., Maabout, S. (eds.) MEDI 2013. LNCS, vol. 8216, pp. 212–224. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41366-7_18
17. Jami, M., Ireland, A.: A verification condition visualizer. In: Giannakopoulou, D., Kroening, D. (eds.) VSTTE 2014. LNCS, vol. 8471, pp. 72–86. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12154-3_5
18. Eriksson, J.: Tool-supported invariant-based programming. Ph.D. thesis, Turku Centre for Computer Science, Finland (2010). http://urn.fi/URN:ISBN:978-952-12-2446-1

Robotics and Integrated Formal Methods: Necessity Meets Opportunity

Marie Farrell(✉), Matt Luckcuck, and Michael Fisher

Department of Computer Science, University of Liverpool, Liverpool, UK
{marie.farrell,m.luckcuck,mfisher}@liverpool.ac.uk

Abstract. Robotic systems are multi-dimensional entities, combining both hardware and software, that are heavily dependent on, and influenced by, interactions with the real world. They can be variously categorised as embedded, cyber-physical, real-time, hybrid, adaptive and even autonomous systems, with a typical robotic system being likely to contain all of these aspects. The techniques for developing and verifying each of these system varieties are often quite distinct. This, together with the sheer complexity of robotic systems, leads us to argue that diverse formal techniques must be integrated in order to develop, verify, and provide certification evidence for, robotic systems. Furthermore, we propose the fast evolving field of robotics as an ideal catalyst for the advancement of integrated formal methods research, helping to drive the field in new and exciting directions and shedding light on the development of large-scale, dynamic, complex systems.

1 Introduction

Formal methods are used in a variety of domains to establish the correctness of both hardware and software systems. Integrating formal methods so that they may be used in a complementary fashion continues to be a difficult challenge that is only exacerbated by the plethora of languages, logics, theorem provers, and model-checkers available. In this paper, we propose robotic systems as an ideal candidate for the large-scale application of integrated formal methods. In fact, it is a fast evolving area where only *integrated* formal methods will suffice. Further, the application of integrated formal methods in the robotics domain will enhance integrated formal methods research and promote their adoption for other large-scale, engineered systems.

Robotic systems are complex and multi-dimensional, with a wide range of concerns: software, hardware, human control, autonomous agent control, reconfigurability, etc. They present numerous challenges for formal verification such as modelling a dynamic environment, providing sufficient evidence for certification and trust, modelling multi-robot systems and, ensuring that autonomous robots

Work supported through EPSRC Hubs for Robotics and AI in Hazardous Environments: EP/R026092 (FAIR-SPACE), EP/R026173 (ORCA), and EP/R026084 (RAIN).

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 161–171, 2018.
https://doi.org/10.1007/978-3-319-98938-9_10

are safely reconfigurable and their decisions do not have dangerous side-effects. We discuss each of these challenges in Sect. 2 as well as the current approaches to tackling them. Our position is that the use of *integrated* formal methods can mediate these difficulties. In Sect. 3, we illustrate the benefits of integrated formal methods with respect to these challenges and outline potential future directions for this research. Finally, Sect. 4 provides concluding remarks. The work cited here is not a complete list, it is drawn from a larger survey of formal specification and verification approaches for autonomous robotic systems[1], which is still in progress.

2 Formal Approaches to Robotic Challenges

This section discusses some of the most crucial challenges to the formal verification of robotic systems and how current formal techniques approach them. First, we discuss the verification of a robotic system's interaction with an unknown and dynamic environment (Sect. 2.1). A further challenge is ensuring that verification methods can provide suitable evidence either for certification or to gain public trust (Sect. 2.2). Certain types of robotic systems present specific challenges and we discuss the challenges posed by modelling multi-robot systems in Sect. 2.3. Finally, Sect. 2.4 describes the challenges when verifying an autonomous or reconfigurable robotic system.

Other challenges include the formal refinement of robotic system specifications to implementable code and ensuring that this final implementation corresponds to its specification in a provably correct way. The heterogeneous nature of robotic systems where various programming languages are used in the implementation of distinct components of the system means that this final specification-to-code step is not trivial. Since humans interact with these systems, it is also necessary to model the human component of the system, however, the verification of human behaviour is largely beyond the reach of current formal verification approaches [34,35].

2.1 Modelling the Physical Environment

To ensure that a robotic system can cope in real-world scenarios, it must be able to react appropriately to an unknown and dynamic environment. When formally modelling robotic systems, the environment is often ignored [19] or assumed to be static and known, prior to the robot's deployment [17,23,34], which is often neither possible nor feasible in the real world. Other approaches abstract away from the environment and rely on predicates representing sensor data about the environment [13]. This insulates the high-level control from the detailed environment, but leaves issues such as sensor and actuator correctness to be dealt with. Formal models of a robotic system's environment must bridge the *reality gap*, the difficulty of transferring models of the environment to the

[1] http://tiny.cc/Luckcuck2018.

real world [13]. This is especially problematic when real-world interactions can impact safety. Reducing the impact of the reality gap often produces intractable models which makes the verification task particularly difficult [11].

Two popular approaches are to either model or monitor the physical environment. Temporal logics have been used to model robotic systems' environments. For example, safety rules and the environment of a robotic assistant captured in Probabilistic Temporal Logic (PTL) [34]. However, as the rules and environment become more complex this approach may not be feasible due to the current limitations of model-checking techniques, unless the properties to be verified can be simplified.

Specifying a monitor to restrict the robotic system to safe behaviours within its environment reduces the verification burden, as only the monitor needs to be verified [21]. For example, a robot's environment can be captured by timed automata and safety properties written in temporal logic [2]. This can be used to build a run-time monitor for the safety properties. This combination of verification methods can help to handle the dynamic environment.

Navigating an unknown and dynamic environment is a challenging task for robotic systems and a number of navigation algorithms exist. However, not all can be employed in safety-critical scenarios as they have not been verified [25]. This suggests that there are limitations of current formal methods to verify these algorithms, and also leads to hybrid approaches that have high computational complexity. KeYmaera is a hybrid theorem prover that has been used to verify both the discrete and continuous behaviour of robotic vehicle navigation using differential dynamic logic for hybrid systems [22].

2.2 Trust and Certification Evidence

Robotic systems are often safety-critical, such as those in nuclear or aerospace applications, and so require certification. Other robotic systems operate in unregulated areas that require public trust, such as domestic assistants. Emerging robotic systems, like autonomous vehicles, require both certification and public trust. Therefore, ensuring that formal verification of robotic systems can provide appropriate trust and certification evidence is crucial. Generally, robotic systems development provides insufficient evidence for certification and public trust, which can hamper their adoption [34]. This is an area where integrating formal methods with current non-formal engineering techniques may prove fruitful.

Further to extensive testing, safety cases are generally used to provide evidence for certification bodies. A safety case is a structured argument that is supported by a collection of evidence providing a compelling, comprehensible and valid case that a system is safe for a given application in a specific environment [9]. Automating the generation of such documentation is a challenging task that must account for heterogeneous content such as physical formulae from the design of the physical system, maintenance procedures, and software (which itself, may be of a heterogeneous nature). Recent work in this area includes a

methodology for automatically generating such safety cases for the Swift pilotless aircraft system using a domain theory and AUTOCERT [9].

Formal methods can provide suitable evidence for certification. For example, Isabelle/HOL and temporal logic have been used to formalise a subset of traffic rules for vehicle overtaking [29]. Furthermore, a model-checking approach has been used to capture the rules and expectations of pilots in order to to provide certification evidence for an autonomous pilotless aircraft [33]. Here, it is verified that the agent controlling the aircraft (in place of a pilot) preserves the rules and recommendations specified by the Civil Aviation Authority (captured as temporal logic formulae).

There are currently no guidelines to help developers choose the most suitable formal method to verify their system [20]. Similarly, regulators and certification bodies are often hesitant to suggest suitable formal methods for safety-critical systems – though guidance has started to appear more recently. Regulators, developers, and academia thus face the challenge of how to determine suitable and robust formal methods for particular types of robotic system.

2.3 Multi-robot Systems

Historically, the development of multi-robot systems has taken inspiration from biological systems such as swarms of insects. Robot swarms are difficult to develop because they are programmed at the microscopic level (that of individual robots) but are intended to exhibit emergent, macroscopic behaviour (at the level of the whole swarm). They are often developed bottom-up, using trial and error to form a swarm with the desired emergent behaviour [23]. Ensuring that macroscopic behavioural requirements (or restrictions) are implemented (or obeyed) at microscopic level can be difficult because of their different abstraction levels.

Robot swarms can be quite large, and so a challenge when verifying robot swarms using current model-checking techniques is state space explosion caused by the large number of concurrent, interacting agents and the system's dynamic environment [1]. This can be mitigated by making use of the homogeneity of the swarm's robots, for example by exploiting symmetry reduction [3] or abstracting the swarm to a single state machine with a counting abstraction [19]. However, both of these approaches only consider swarms with homogeneous behaviour.

The emergent behaviour of robot swarms can be captured using temporal logic [38] and often lend themselves to probabilistic models. In particular, PRISM has been used to encode probabilistic state machines which can then be checked for properties specified in a probabilistic temporal logic [19].

Further, robotic systems may consist of a team of heterogeneous robots. In hazardous environments such as nuclear plants, it is conceivable that multiple robots may be required in order to complete a specific task. Each robot would have a distinct role, for example, a robotic arm to examine a piece of debris and a more mobile robot to monitor, calibrate and mend the robotic arm should it malfunction. The robot team can be verified at the macroscopic level, but at the microscopic level each robot must be verified individually. For example, one robot

in the team may be characterised by a verifiably correct Z specification, whereas another may be verified by model-checking its source code. The behaviour of the robot team might be different to the behaviour of each individual robot and thus another approach may be used to verify the team's behaviour. It is not clear how best to link the verification approaches taken at these different levels of abstraction because the approaches most amenable to the verification of each individual robot might be different.

2.4 Adaptation, Reconfigurability and Autonomy

A self-adaptive system continually alters its behaviour in a feedback loop that is driven by its environment. A literature survey found that there are no standard tools for the formal modelling and verification of self-adaptive systems [36]. Of the tools surveyed, 30% use model-checking. One avenue of research suggests using (both semi-formal and formal) models to check run-time behaviour [6]. This agenda considers approaches such as automatic test case generation and formal model-checking. The aim being to reduce state explosion by quantifying as many variables as possible at run-time.

Related to this is the notion of a reconfigurable system, which senses its environment and makes a *decision* about how best to reconfigure itself to suit changes in its requirements or the environment. Reconfiguration is essential for ensuring the fault tolerance of safety-critical robotic systems [32]. There are two key open questions when applying formal methods to these systems: (1) how to specify and analyse a configuration, and (2) how to compare different configurations of the same system [24]? The design of reconfigurable hardware has received much attention, but autonomous software reconfiguration remains a challenge [4]. One approach involves building a flexible control system that can reconfigure itself once a fault is detected [5]. Z models can be used to describe an arbitrary reconfigurable system [37]. The model provides a method for describing and comparing different configurations of the system's architecture.

Since reconfigurability requires the system to make an autonomous decision as to how best to reconfigure itself, it is vitally important that the decisions made by the system are *rational*, meaning that the system can explain its reasoning. This leads us to model the motivations and decisions of the system, ideally as first class objects [13].

Agent-based systems are one way of describing autonomy; there are many different models of agent systems, based on different models of autonomy. Agents are used to model a robot's interactions with other actors, its environment, and the physical environment itself. For example, probabilistic temporal logics have been used for modelling an autonomous mine detector robot, controlled by an agent, and its environment [16]. The model of the agent can be used for both design- and run-time verification.

A formal model of a style of agent system using Z has been devised that gives its agents a formal semantics [12]. The interactions of multiple interacting agents can also be modelled using finite state machines. These are converted into

Alloy specifications for automatic verification [26]. The size of these specifications meant that keeping the models tractable was challenging.

Relating agent programs, written in an agent programming language, and agent (verification) logics remains an open problem. One approach has been to define an agent programming theory combining an agent programming language and verification logic [15]. Program model-checking such as the Model-Checker for Multi-Agent Systems (MCMAS), has been used to verify heterogeneous agents interacting with an environment [7]; and Agent Java PathFinder (AJPF) [10], which can model-check programs written in a particular style of agent language.

3 Integrated Formal Approaches to Robotic Challenges

In Sect. 2, we outlined the challenges encountered when developing reliable robotics and a number of current (non-integrated) approaches to addressing them. It is clear that only by using a combination of specialised tools and methodologies can we achieve a high level of confidence in software. For example, the NASA Remote Agent uses specialized languages for each of the planner, executive, and fault diagnosis components [30].

There is currently no general framework integrating formal methods for robotic systems. However, Sect. 3.1 describes recent trends and some notable bespoke examples of integrated formal methods (iFM) for robotics. In Sect. 3.2 we discuss how robotics and iFM can benefit from one another. We do not ignore the important role that validation techniques, such as testing and simulation, play in the development of robotic systems. These too, should be integrated into the development process to be used alongside formal methods [35].

3.1 Adopting iFM for Robotics

Current approaches to formal verification in robotics typically centre around one tool or technique that is suited to verifying properties of a particular type (concurrency, probability, etc.). It is clear from the increasingly complex nature of robotic systems that this is not a sustainable approach to ensuring the correctness of these systems. These approaches suffer from a number of drawbacks that are mostly caused by the limitations of their logic or the tool being used.

A comparison of four different specification formalisms (CSP, WSCCS, Unity Logic, and X-Machines) for specifying and verifying emergent swarm behaviour, concluded that a blending of these formalisms offered the best approach to specify emergent swarm behaviour as none was sufficient in isolation [14]. This claim is further supported by the use of MAZE (an extension of Object-Z for multi-agent systems) that uses Back's action refinement to facilitate a top-down development process of the swarm, from the macroscopic to microscopic level [31].

It is therefore clear that the use of iFM can help to mediate the issues surrounding the development of robotic systems as has been illustrated by the following bespoke examples. We outlined the importance of reconfigurable systems

in Sect. 2.4, and a combination of Event-B and the PRISM model-checker has been used to derive a reconfigurable architecture for an on-board satellite system [32]. The combination of these formal notations allows not only for the formal specification and derivation (via refinement) of the system in Event-B, but also the probabilistic assessment of its reliability and performance using PRISM.

The combination of AJPF for agent verification, Uppaal for timing properties and, spatial reasoning has been used to verify the procedures for a driverless car joining and leaving a vehicle platoon [17,18]. This work verifies the cooperation between the vehicles, and the abstract behaviour of the real physical vehicle. Related work uses CSP||B to correctly model a real physical platooning vehicle [8].

Finite State Processes (FSP) and πADL (π-calculus combined with the Architecture Description Language) have been combined to capture safety and liveness properties of multi-agent robotic systems [1]. The FSP specifications of the relevant safety and liveness properties are transformed into Labelled Transition Systems, then the agent programs and architecture (described in πADL) are checked to see if they satisfy the required properties. Designed as a generic notation for modelling robotic systems, RoboChart integrates the process algebra CSP with a graphical timed state machine notation [28]. This allows graphical visualisation of the specification and automatic model-checking of its behaviour. The use of a process algebra in these cases is ideal for modelling communication across a channel with πADL and timed state machines, respectively, providing a robust model of the system's state that could not be achieved using a process algebra in isolation.

Furthermore, several views of the same system or component often require integration of analysis, even for one specification element. For example, model-checking, model-based testing, and user evaluation have all been applied to the same robotic system [35]. Each, however, works at a very different level of abstraction and formality and so the challenge here is to integrate this breadth of techniques in a holistic framework.

3.2 Future Directions for iFM

The benefits of iFM are well known. Specifically with respect to robotics, iFM can: (1) enable us to capture detailed physical environments by combining static and dynamic models; (2) provide a formal mechanism for linking the macroscopic and microscopic levels of multi-robot systems; (3) provide robust evidence for trust and certification, and; (4) express the complex properties of adaptive, reconfigurable, and autonomous systems. This unique set of challenges posed by robotics provides tangible targets for iFM researchers. Thus, robotics can benefit from and be a catalyst for iFM research, and the adoption of these techniques for large-scale, dynamic, and complex systems.

Integrating multiple approaches to verification for systems in the safety-critical domain presents its own set of familiar challenges, such as increased complexity and ensuring the correctness of the integrated model. Until now, these challenges have generally been addressed using theoretical frameworks, small

case studies, and prototype tools. Robotic systems are a complex and practical field where iFM is crucial to provably correct advances and adoption. Further to these challenges, usability is a concern from both perspectives. Firstly, the iFM community is tasked with providing a set of robust tools that are intuitive and usable for the developers of robotic systems. Secondly, robotic systems should be developed with iFM in mind using a set of standardised, modular constructs that are amenable to iFM.

Combining formal methods with different strengths and weaknesses, such as exhaustive model-checking and proof-based methods, provides a useful balance of complexity and robustness. While model-checking exhaustively examines the system's state space to check if a property is preserved, formal proofs provide a step-by-step mathematical argument as to why the property holds. Although, both model-checking and theorem proving generally involve abstracting to a formal specification in order to verify the system, an advantage of model-checking is that it can be used directly on the implemented code, whereas theorem proving cannot. In contrast, theorem proving techniques do not suffer from the explosion in state space that limits the complexity of the properties that can be verified using model-checking. Formalisms that support formal refinement of specifications, such as Event-B, facilitate a verification process that provides a proof of the properties that are verified at each level of abstraction. Integrating model-checking and proof-based approaches to verification will provide fast identification of bugs and a list of the properties that are verified using model-checkers, as well as robust mathematical arguments for correctness in the form of proofs. These techniques, in combination, can thus provide the more robust certification evidence required for robotic systems.

Robotic systems are layered entities containing both hardware and software components. In general, each layer is built upon the lower layers and assumes that they behave correctly. In this scenario it is likely that the formalisms and tools used for both the verification and implementation of each layer are different. This presents a huge challenge for ensuring the correctness of the entire system, and in particular, verifying the interactions between these layers. Contemporary robotics software is often highly modular, with components loosely connected. For example the Robot Operating System (ROS) allows architectures comprising heterogeneous components, written in different programming languages, which can interface with a range of hardware and software components [27]. There will undoubtedly be different techniques relevant to, and optimal for, the verification and validation of each of the different components. These include stochastic analysis of a learning component, a range of testing techniques for a vision component, or model-checking of an autonomous decision-maker. These must all be combined to provide a coherent and comprehensive analysis of the whole system.

We propose the verification of middleware architectures, such as ROS, as an ideal starting point for this research agenda. For example, the specification and verification of individual ROS nodes using pre- and post-conditions or assume-guarantee clauses, written using a heterogeneous collection of logics, would prove

useful here. This approach could also aid in the verification of heterogeneous teams of robots as discussed in Sect. 2.3. It is clear that a common framework for translating between, relating, or integrating different formal methods and validation techniques will prove useful. Such a framework would enable easy conversion between formalisms and verification tools. This would facilitate the use of heterogeneous models that are each suited to a particular type of behaviour or property. Moreover, the use of iFM can save time in the development process by avoiding duplicate specifications and exploiting different types of verification tools for proving different properties of the same system.

4 Conclusions

Robotic systems are inherently multi-dimensional entities that combine both hardware and software components that interact with humans and the physical world. These systems can be modelled in a variety of ways and thus must integrate verification and validation techniques from the fields of embedded, cyber-physical, real-time, hybrid adaptive and even autonomous systems. In Sect. 2, we discussed the challenges that are encountered when developing certifiably correct robotic systems and the current formal approaches to tackling them. It is clear that current (non-integrated) formal methods are not robust enough, particularly in isolation, to ensure the correctness of these systems. Although not without its challenges, in Sect. 3, we have illustrated the benefits of employing *integrated* formal methods in this setting and outlined future directions for this work. Furthermore, we have argued that although robotics actually necessitates the use of integrated formal methods, integrated formal methods can utilise robotics as a viable and impactful means for advancing integrated formal methods research and their adoption for large-scale, complex systems. The challenges that we have outlined throughout this paper can be achieved through funding streams such as EPSRC at a national level and Horizon 2020 at an international level.

References

1. Akhtar, N.: Contribution to the formal specification and verification of a multi-agent robotic system. Eur. J. Sci. Res. **117**(1), 35–55 (2014)
2. Aniculaesei, A., Arnsberger, D., Howar, F., Rausch, A.: Towards the verification of safety-critical autonomous systems in dynamic environments. Electron. Proc. Theor. Comput. Sci. **232**, 79–90 (2016)
3. Antuña, L., Araiza-Illan, D., Campos, S., Eder, K.: Symmetry reduction enables model checking of more complex emergent behaviours of swarm navigation algorithms. In: Dixon, C., Tuyls, K. (eds.) TAROS 2015. LNCS (LNAI), vol. 9287, pp. 26–37. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22416-9_4
4. Bi, Z.M., Lang, S.Y.T., Verner, M., Orban, P.: Development of reconfigurable machines. Int. J. Adv. Manuf. Technol. **39**(11–12), 1227–1251 (2008)
5. Braman, J.M.B., Murray, R.M., Wagner, D.A.: Safety verification of a fault tolerant reconfigurable autonomous goal-based robotic control system. In: International Conference on Intelligent Robots and Systems, pp. 853–858. IEEE (2007)

6. Cheng, B.H.C., et al.: Using models at runtime to address assurance for self-adaptive systems. In: Bencomo, N., France, R., Cheng, B.H.C., Aßmann, U. (eds.) Models@run.time. LNCS, vol. 8378, pp. 101–136. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08915-7_4
7. Choi, J., Kim, S., Tsourdos, A.: Verification of heterogeneous multi-agent system using MCMAS. Int. J. Syst. Sci. **46**(4), 634–651 (2015)
8. Colin, S., Lanoix, A., Kouchnarenko, O., Souquières, J.: Using CSP||B components: application to a platoon of vehicles. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 103–118. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03240-0_11
9. Denney, E., Pai, G.: Automating the assembly of aviation safety cases. IEEE Trans. Reliab. **63**(4), 830–849 (2014)
10. Dennis, L.A., Fisher, M., Webster, M., Bordini, R.H.: Model checking agent programming languages. Autom. Softw. Eng. **19**(1), 5–63 (2012)
11. Desai, A., Dreossi, T., Seshia, S.A.: Combining model checking and runtime verification for safe robotics. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 172–189. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_11
12. D'Inverno, M., Luck, M., Georgeff, M., Kinny, D., Wooldridge, M.: The dMARS architecture: a specification of the distributed multi-agent reasoning system. Auton. Agent. Multi. Agent. Syst. **9**(1/2), 5–53 (2004)
13. Fisher, M., Dennis, L.A., Webster, M.: Verifying autonomous systems. Commun. ACM **56**(9), 84–93 (2013)
14. Hinchey, M.G., Rouff, C.A., Rash, J.L., Truszkowski, W.F.: Requirements of an integrated formal method for intelligent swarms. In: Formal Methods for Industrial Critical Systems, pp. 125–133. ACM Press (2005)
15. Hindriks, K.V., Meyer, J.-J.C.: Toward a programming theory for rational agents. Auton. Agent. Multi. Agent. Syst. **19**(1), 4–29 (2009)
16. Izzo, P., Qu, H., Veres, S.M.: A stochastically verifiable autonomous control architecture with reasoning. In: IEEE Conference on Decision and Control, pp. 4985–4991 (2016)
17. Kamali, M., Dennis, L.A., McAree, O., Fisher, M., Veres, S.M.: Formal verification of autonomous vehicle platooning. Sci. Comput. Program. **148**, 88–106 (2017)
18. Kamali, M., Linker, S., Fisher, M.: Modular verification of vehicle platooning with respect to decisions, space and time. arXiv preprint arXiv:1804.06647 (2018)
19. Konur, S., Dixon, C., Fisher, M.: Analysing robot swarm behaviour via probabilistic model checking. Robot. Auton. Syst. **60**(2), 199–213 (2012)
20. Kossak, F., Mashkoor, A.: How to select the suitable formal method for an industrial application: a survey. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 213–228. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_13
21. Machin, M., Dufossé, F., Blanquart, J.-P., Guiochet, J., Powell, D., Waeselynck, H.: Specifying safety monitors for autonomous systems using model-checking. In: Bondavalli, A., Di Giandomenico, F. (eds.) SAFECOMP 2014. LNCS, vol. 8666, pp. 262–277. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10506-2_18
22. Mitsch, S., Ghorbal, K., Platzer, A.: On provably safe obstacle avoidance for autonomous robotic ground vehicles. In: Robotics and Science and Systems (2013)
23. Moarref, S., Kress-Gazit, H.: Decentralized control of robotic swarms from high-level temporal logic specifications. In: International Symposium on Multi-robot and Multi-agent Systems. IEEE (2017)

24. Morse, J., Araiza-Illan, D., Lawry, J., Richards, A., Eder, K.: Formal specification and analysis of autonomous systems under partial compliance. arXiv preprint arXiv:1603.01082 (2016)
25. Phan, D., Yang, J., Ratasich, D., Grosu, R., Smolka, S.A., Stoller, S.D.: Collision avoidance for mobile robots with limited sensing and limited information about the environment. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 201–215. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_13
26. Podorozhny, R., Khurshid, S., Perry, D., Zhang, X.: Verification of multi-agent negotiations using the alloy analyzer. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 501–517. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73210-5_26
27. Quigley, M., et al.: ROS: an open-source robot operating system. In: ICRA Workshop on Open Source Software, vol. 3, p. 5 (2009)
28. Ribeiro, P., Miyazawa, A., Li, W., Cavalcanti, A., Timmis, J.: Modelling and verification of timed robotic controllers. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 18–33. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_2
29. Rizaldi, A., et al.: Formalising and monitoring traffic rules for autonomous vehicles in Isabelle/HOL. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 50–66. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_4
30. Simmons, R., Pecheur, C., Srinivasan, G.: Towards automatic verification of autonomous systems. In: International Conference on Intelligent Robots and Systems, vol. 2, pp. 1410–1415. IEEE (2000)
31. Smith, G., Li, Q.: MAZE: an extension of Object-Z for multi-agent systems. In: Ait Ameur, Y., Schewe, K.D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 72–85. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_6
32. Tarasyuk, A., Pereverzeva, I., Troubitsyna, E., Latvala, T., Nummila, L.: Formal development and assessment of a reconfigurable on-board satellite system. In: Ortmeier, F., Daniel, P. (eds.) SAFECOMP 2012. LNCS, vol. 7612, pp. 210–222. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33678-2_18
33. Webster, M., Cameron, N., Fisher, M., Jump, M.: Generating certification evidence for autonomous unmanned aircraft using model checking and simulation. J. Aerosp. Inf. Syst. **11**(5), 258–279 (2014)
34. Webster, M., et al.: Toward reliable autonomous robotic assistants through formal verification: a case study. IEEE Trans. Hum.-Mach. Syst. **46**(2), 186–196 (2016)
35. Webster, M., et al.: An assurance-based approach to verification and validation of human-robot teams. arXiv preprint arXiv:1608.07403 (2016)
36. Weyns, D., Iftikhar, M.U., de la Iglesia, D.G., Ahmad, T.: A survey of formal methods in self-adaptive systems. In: International C* Conference on Computer Science and Software Engineering, pp. 67–79. ACM (2012)
37. Weyns, D., Malek, S.: FORMS: a formal reference model for self-adaptation. In: International Conference on Autonomic Computing, pp. 205–214. ACM (2010)
38. Winfield, A.F., Sa, J., Gago, M.C.F., Dixon, C., Fisher, M.: On formal specification of emergent behaviours in swarm robotic systems. Int. J. Adv. Robot. Syst. **2**(4), 363–370 (2005)

Formal Modelling of Software Defined Networking

Vashti Galpin(✉)

Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh, Edinburgh, UK
Vashti.Galpin@ed.ac.uk

Abstract. This paper investigates the application of Carma, a recently developed quantitative process-algebra-based modelling language, to the stochastic modelling of software defined networking (SDN). In SDN, a single controller (or hierarchy of controllers) determines the behaviour of the switches that forward traffic through the network, and it is used in a variety of settings including cloud and data centres. This research is the initial phase of developing a methodology for agile formal modelling of performance and security aspects of SDN, and focusses on the fat-tree network topology. The results demonstrate that the Carma language and its software tools which include the MultiVeStA statistical model checker provide a good basis for modelling SDN.

1 Introduction

Traditionally, network modelling has been done by emulation on virtual machines using a tool called mininet [29] or simulation using a tool such as ns-3 [9,26]. Both of these approaches consider the full network stack, which can be very expensive in terms of initial setup, as well as computational resources to execute. Other simulation approaches allow for some abstraction [6,8,30] from these details, in order to reduce these overheads and various formal approaches have also been suggested [3,4,7,27,33–36,40,42], most of which have no quantitative features.

Our goal is to provide a novel approach to modelling the behaviour of networks at a moderately high level of abstraction but with the ability to measure performance, something that is missing from most formal approaches for networks. This will still allow for a quantitative assessment of network behaviour which is crucial to evaluate different configurations but provide a lighter-weight approach that the full-stack emulation and simulation methods. Our approach models individual packets traversing a network but abstracts from lower level concerns of the network stack. This may reduce what questions can be answered by the model; however, it will still allow many questions of interest such as packet latency, to be answered much more quickly than the traditional full stack emulation and simulation approaches, and hence provides an alternative approach. As is well known, formal modelling of computer systems has multiple benefits including the ability to reason about a system before it is built, and to conduct experiments using a model of an existing system without disrupting the system itself.

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 172–193, 2018.
https://doi.org/10.1007/978-3-319-98938-9_11

This paper considers how an existing probabilistic modelling language can be used to simulate the behaviour of software defined networking (SDN) [15] at the packet level, allowing for the investigation of performance and security properties, as well as trade-offs between these properties. Specifically, we work with the quantitative formal modelling language CARMA and examine how CARMA and CaSL, the textual language of the CARMA Eclipse Plug-in tool [25,32] support this type of modelling. MultiVeStA [38] is integrated into a command-line version of the tool, allowing statistical model checking of CARMA models on top of simulation.

The textual language of the CARMA tool provides an explicit syntax and a location type for expressing location with respect to a structure that describes discrete space. This motivated the choice of CARMA to model physical network topology and allows for a parametric approach to network topology description. Also importantly for practical application of CARMA, the tool also provides a rich choice of attribute types, including integer, real, enumerated types, Boolean, and finite lists and sets of these types. Functions can be defined over all data types, to support programmatic aspects of models, and goes beyond process-algebra-style behaviour and interaction.

Our experiments show that there is a good match between the discrete space syntax provided by CARMA and modelling network topology. This makes it possible to separate network topology (and traffic) from the definition of generic network elements such as hosts, switches and controllers. This has advantages in terms of speed of model construction as well as ease of debugging models. The three major contributions of this research are as follows. First, it provides an assessment of CARMA for modelling network performance and security in SDN through the development of a model that permits packet-level modelling. The model contains generic controller, switch and host components that allow for the controller to send flow table rules to switches which are then able to direct packets from one host to another through the network. Packets are modelled explicitly and their header content is used by switches to determine how they should be handled depending on the flow rule that applies. Furthermore, the model is parametric with respect to the network description, allowing fast development of models with different topologies.

Second, it allows for experimentation with the fat-tree topology in a SDN setting that considers the scalability of the topology with respect to packet latency which is a standard measure of network performance, considering both uniform and MapReduce traffic; with the goal of determining the packet rates at which the network become congested and can no longer operate at line speed (the speed of the underlying network connections) because of queues at switches. The performance cost of mitigation of attacks is also considered. Finally, it explores the use of MultiVeStA for statistical model checking of switch queue sizes, allowing for the exploration of the parameter space of MapReduce traffic patterns, thereby integrating the use of different formal methods approaches.

2 Background

This research contributes the first phase of the development of a general methodology for modelling of networks and network security, and it is necessary to select a particular case study for exploration of the potential of the approach. We have chosen to consider the use of software defined networking (SDN) in data centres. This specifies the focus and helps identify suitable examples with which to work. SDN is an ideal setting since it is an industry standard whose deployment is wide-spread and increasing. As networking requirements become more complex in cloud and data centre scenarios, SDN provides a different approach based on a full network overview compared to other approaches. One of the complexities that must be addressed is security and SDN offers opportunities and challenges in this domain [12]. Furthermore, there is an ongoing need for assessment of SDN performance due to the range of implementations and switch types [17].

The distinct roles of network elements in SDN maps well to CARMA components which describe behaviour with the addition of store, allowing for internal state, which is not often a feature of process algebras. Focussing on data centres allows the consideration of regular topologies which is a good starting point for modelling. Regular topologies also make large networking scenarios possible programmatically and we will show later in the paper how CARMA supports this.

2.1 Software Defined Networking

In traditional networking, routers direct packets and have enough knowledge about the state of the network to make forwarding decisions. Software defined networking [15] takes a very different approach whereby the network switches are provided with flow rules (by the controller which has an overview of all network behaviour) that specify how packets should be directed. Each switch has a flow table that is stored in fast (but expensive) ternary content-addressable memory (TCAM) which allows for fast look-up. When a packet arrives at a switch, its header is compared with the flow table entries. These entries may contain wildcards, and different packet headers may match a single flow rule. If a match is found, the action specified by the rule is followed and counters for the rule are updated, and if not the packet header is sent to the controller (which may add a new rule to the switch for that packet header). The two most common actions for a rule are **forward** on a port number, or **drop** where the packet is not forwarded.

Rules in the flow table can be divided into proactive and reactive. Reactive rules are those that are installed when the controller must decide what to do with a packet that does not match the rules at a switch. By contrast, proactive rules are those that are installed by the controller as a switch becomes active, based on an overview of the network topology and specific choice of a single route between each pair of hosts. We focus on proactive rules and the performance of a balanced routing over a network. This is not a limitation of the modelling as a variant of this model has been used to consider reactive rules in the evaluation of an attack mitigation [10].

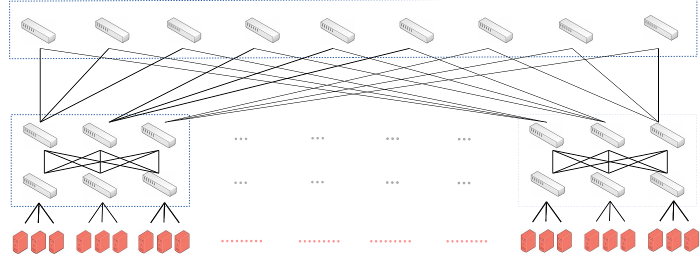


Fig. 1. Fat tree with 6-port switches

The controller makes decisions programmatically about the flows through the network, determined routes from information about network topology, existing traffic and updating routes when necessary. This route choice can have different aims such as performance (efficient use of network bandwidth) or security (for example, mitigation for covert channel attacks [41]).

2.2 Fat-Tree Topology

Our focus is on the fat-tree topology [2] that is used in data centres and is suitable for use with SDN. In a standard tree topology, there may be a single switch at the top level of the tree (the core) and this is a bottleneck. By contrast, the fat-tree topology is based on k-port switches and provides more than one core switch. Figure 1 illustrates the fat-tree topology obtained using 6-port switches The top block of nine switches are the *core*. There are then k pods, each containing k switches. The layer of switches immediately under the core is referred to as the *aggregate* and the layer before the hosts, the *edge*. The 6-port topology supports 54 hosts and there are 9 routes between each pair of hosts that are in different pods, as can be seen from the figure by considering the first host of the first pod and the first host of the last pod. The controller is not included in this diagram and there is research into where controllers should be placed for best performance [22]. In this research, we abstract from these details, as discussed later.

More generally, a 3-level fat tree based on commodity switches with k ports has a core of $k^2/4$ switches and k^2 switches at the aggregate and edge level. This allows for the support of $k^3/4$ hosts. Between each pair of hosts, there are $k^2/4$ routes, one through each switch of the core. The use of commodity switches allows for a cheap but efficient topology, and it is well suited for SDN because of multiple routes.

Over and above modelling topologies, we need to model traffic patterns as well. In the experiments, we will consider two traffic patterns: one where there is traffic between all hosts where we consider at what traffic levels, the packet latency becomes too high. The second considers the MapReduce pattern where many hosts communicate with a single host to convey the results of calculations done in parallel.

3 Related Work

In the case of SDN, mininet [29] allows for emulation by modelling actual network behaviour on multiple virtual machines. A limitation of mininet is that it runs in real-time as an emulator and hence does not scale to large systems. Network simulators such as ns-3 [9,26] simulate the full network stack behaviour. Both of these approaches are costly in terms of initial set-up and have steep learning curves. An alternative is a much more abstract approach such as CloudSim which is used to model large SDN data centres [6] but this is an understanding of network performance before building the model, and hence is not suitable for the type of network performance and security modelling proposed here. A different approach is taken using TopoGen in modelling of SDN topologies [30] where the focus of the tool is in supporting large topologies and then using hybrid modeling, simulation and control of data networks based on a hybrid DEVS (Discrete Event System Specification) formalism where some packets are modelled explicitly and some as flows [8].

Various formal approaches have been suggested such as NetKat [4], Veriflow [27] and others [3,7,33–36,40,42]. Probabilistic NetKat [39] is the closest to our approach but is limited to a time-homogeneous approach. Using Carma, it is possible to consider behaviour over time as parameters vary, allowing for the dynamic modelling of the effects together with mitigation of attacks, as illustrated later. Some process algebras have also been proposed for network modelling such as [28] but most focus on wireless or ad hoc networks and are not quantitative such as [16]. In the case of the spatial extension of PEPA (which is a stochastic process algebra that influenced the development of Carma) [18], Carma offers a much richer way to specify behaviour.

4 Overview of Carma

Carma (Collective Adaptive Resource-sharing Markovian Agents) is a process-algebra-based quantitative modelling language developed for the modelling of collective adaptive systems with explicit support for the modelling of discrete space, as well as separate specification of an environment. [25,32]. It has roots in process algebras developed for performance evaluation such as PEPA [23] and biological modeling such as Bio-PEPA [11], and has (time-inhomogeneous) continuous-time Markov chain semantics. It has richer interaction than PEPA or Bio-PEPA, and uses attribute-based communication similar to that of SCEL [14] and AbC [1]. It has been used to model taxi movement [24], carpooling [43], ambulance deployment [19] and pedestrian mobility [21].

The language has two forms: the mathematical definition of Carma [25] and the language CaSL [20] of the Carma Eclipse Plug-in (available at quanticol.sourceforge.net). Both will be used in this paper. The former will be used to provide abstract presentations of the models, and the latter to describe how measures and spatial concepts are defined with the software tools, and thereby illustrate the power of the implementation with respect to network topology modelling.

The basic behavioural unit of a CARMA model is a *component* which consists of communicating behaviour specified using process-algebra-style prefixes (actions), an initial behaviour and a store of attributes that characterise the component. Interaction between components can be unicast or broadcast. The components of a model form a *collective* which then operates within an *environment*. The environment includes a global store, and updates to elements in the store are triggered by actions performed by components. It also specifies the rates and probabilities at which actions are performed, and allows for new components to be added to the collective when given actions are performed by components. In the context of the SDN model, packet rates will appear in the environment as will global attributes to calculate packet latency.

To understand the basic behaviour of component, consider a component that has three attributes v, x and y in its store. Behaviour in this component can be specified in the following form. Here $\top$ indicates true and $\bot$ false.

$$\begin{aligned} A \stackrel{\text{def}}{=}\ & \mathsf{signal}[\top]\langle v\rangle\{a \leftarrow a-1\}.A\ + \\ & \mathsf{new_signal_count}^{\star}[\mathsf{my}.b < b](new_a)\{a \leftarrow new_a\}.A\ + \\ & [a=0]\mathsf{finished}^{\star}[\bot]\langle\rangle.\mathbf{nil} \end{aligned}$$

Process A can repeatedly send out a signal of v to one other component (and wait until it is received) at which point the value of a is decreased by one. It can also receive a broadcast communication (indicated by the asterisk) from any other component which has a larger b value than it, which communicates a value that is then stored as a. This is attribute-based communication: A can only "hear" from components with a larger b value. However, if a becomes 0, then the process can perform an internal action and become the process with no behaviour. The use of a broadcast action with a false predicate leads to an internal action since no other component can satisfy the predicate, and broadcast is not blocking. Thus the action happens without interaction. As will be seen in the SDN model, sometimes an attribute is updated by calling a function, and no interaction with other components is necessary, and hence we use this form of action.

The component containing behaviour A, say $CompA$, can be described as a component that attempts to communicate (by unicast) the value v a certain number of times after which it ceases communication. However, imagine that there are other components (not specified here) which are allowed to communicate a new value of a to $CompA$. This can either reduce or increase how many more times the value v is sent by $CompA$. The other components which can communicate new values must have larger b values (which could be a priority) than $CompA$ to successfully change a.

Formally, components have the syntax $C ::= \mathbf{0} \mid (P, \gamma)$ where $\mathbf{0}$ is the null component, P is a process that describes behaviour and γ is the store. Stores map from *attribute names* to *basic values*. The syntax of processes that define the behaviour of components are specified by the following

$$
\begin{array}{rlll|l}
P, Q ::= & \textbf{nil} \mid \textbf{kill} & & & act ::= \alpha^{\star}[\pi]\langle\overrightarrow{e}\rangle\sigma \mid \alpha\ [\pi]\langle\overrightarrow{e}\rangle\sigma \\
\mid & act.P \mid P + Q & & & \quad \mid \alpha^{\star}[\pi](\overrightarrow{x})\sigma \mid \alpha\ [\pi](\overrightarrow{x})\sigma \\
\mid & P \mid Q \mid [\pi]P & & & e ::= a \mid \mathsf{my}.a \mid x \mid v \mid \mathsf{now} \mid \cdots \\
\mid & A & (A \stackrel{\text{def}}{=} P) & & \pi ::= \top \mid \bot \mid e_1 \bowtie e_2 \mid \neg\pi \mid \pi \wedge \pi \mid \cdots
\end{array}
$$

where

- α is an *action type*; π is a *predicate*;
- e is an *expression*; x is a *variable*; $\overrightarrow{\cdot}$ indicates a sequence of elements;
- a is an *attribute name*; v is a *basic value*;
- σ is an *update* defined by a function from Γ to $Dist(\Gamma)$ where $Dist(\Gamma)$ is the set of probability distributions over Γ. This allows for stochastic updates.

The behaviour includes the absence of behaviour **nil**, the ability to remove a component from the collective **kill**, action prefix $act.P$, choice $P + Q$, parallel composition $P \mid Q$, predicate prefix $[\pi]P$ where the behaviour as P is only available if the guard π defined over the component's attributes is true, and constant definition. Expressions include now for the current simulation time and $\mathsf{my}.a$ which refers to the value of the attribute a in the current component. When referring to an attribute shared by a two components, $\mathsf{my}.a$ allows for distinction between the two in predicates that constrain interaction.

The different prefixes specify the type of interaction.

Broadcast output: $\alpha^{\star}[\pi]\langle\overrightarrow{e}\rangle\sigma$
Broadcast input: $\alpha^{\star}[\pi](\overrightarrow{x})\sigma$
Unicast output: $\alpha[\pi]\langle\overrightarrow{e}\rangle\sigma$
Unicast input: $\alpha[\pi](\overrightarrow{x})\sigma$

Here α is an action name, π is a predicate over attributes of the sender and the receiver, and σ specifies attribute updates. For output, $\overrightarrow{e}$ is a list of output expressions, and for input, $\overrightarrow{x}$ is a list of variables, as is standard. Broadcast actions are indicated by the presence of an asterisk. As mentioned above, an internal action has the form $\alpha^{\star}[\bot]\langle\rangle\sigma$.

The predicates after the action name in a prefix determine who takes part in the communication. Rates, probabilities and weights associated with an action name are recorded in the environment element of the model and may depend on attributes of the sender (in the case of broadcast which is non-blocking) and on the sender and receiver (in the case of unicast which is blocking). We use predicates in the SDN model to ensure unicast communication only occurs between components that are directly connected in the network and this is specified by the space description.

A collective N consist of either a component C or the parallel composition of two collectives, $N ::= C \mid N \parallel N$, and a CARMA model then consists of a collective together with an environment $S ::= N \textbf{ in } \mathscr{E}$. The environment collects together all of the information necessary for the collectives to operate including rules that regulate the system such as rates of interaction and probabilities that

interaction may occur, as well as global information. The environment consists of two elements: a *global store* γ_g to record the value of global attributes, and an *evolution rule* ρ. This is a function which, depending on the *current time* (using now), the global store and the current state of the collective returns four functions defined on stores and action names. These are known as the *evaluation context.*

Probabilities: $\mu_p(\gamma_s, \gamma_r, \alpha)$ determines the probability that a component with store γ_r can receive a message from a component with store γ_s when α is executed;

Weights: $\mu_w(\gamma_s, \gamma_r, \alpha)$ determines the weight allocated to α executed by a component with store γ_r receiving a message from a component with store γ_s. This weighting determines the probabilities between different unicast actions.

Rates: $\mu_r(\gamma, \alpha)$ provides the execution rate of action α executed at a component with store γ;

Updates: $\mu_u(\gamma, \alpha)$ determines the updates on the environment (global store and collective) induced by the execution of action α at a component with store γ. The execution of an action can modify the values of global variables and also add new components to the collective.

Figure 6 provides an example of this evaluation context as used in the SDN model. For the rates and updates, the function is expressed as a series of cases based on the particular action involved. For the SDN model, there is little use of the component stores in the definitions. The only explicit occurrence is in the update for the action log_packet*, although there are more occurrences in the elided details of the rates for packet generation and the traffic patterns which could be determined by the identity of the source or destination of a packet.

The operational semantics of Carma specifications are defined through transition relations. The semantic rules can be found in [32]. These relations are defined in the FuTS style [13] and are described using a triple $(N, \ell, \mathscr{N})$ where the first element is a component, or a collective, or a system; the second element is a transition label; and the third element is a function associating each component, collective, or system with a non-negative number. If this value is positive, it represents the rate of the exponential distribution characterising the time needed for the execution of the action represented by ℓ. A zero value is associated with unreachable terms. FuTS style semantics are used because it makes explicit an underlying (time-inhomogeneous) Action Labelled Markov Chain, which can be simulated with standard kinetic Monte Carlo algorithms.

Two further elements are important for modelling with Carma. Measures define the outputs of a Carma model when it is simulated. A space description defines the discrete space model over which a Carma model will operate. It defines a weighted graph structure, and each component can be located at a node in this graph. To aid clarity of presentation, these two elements will be presented using CaSL, the textual language of the software tools, in the next section where the model is introduced.

5 The SDN Model

Carma is a rich formalism developed for a specific purpose; however it is applicable to a variety of systems. For SDN, an important goal of the model is to be parametric with respect to the network specification. Therefore, generic components model various aspects of the system and there is no specification of the network details outside of the portion of the model that describes the network topology and traffic parameters. Thus we need only define four generic components for our model: host, switch, controller and timer. For some scenarios, deterministic time delays are required and the timer component supports their modelling. In the model presented here, it determines when data should be collected from the switches by the controller. Each switch has a unique location and each host is located at a switch.

The interaction between the components is illustrated in Fig. 2 where components with a single border have a single occurrence, and with double borders, multiple occurrences. The Carma components for all four are given in Figs. 3 and 4. The notation x_i refers to element i of array x, except in the case of action names, where action_i refers to indexed action names. The model presented is an abstraction of the CaSL model developed (for reasons of space) and some aspects are only mentioned in passing. Furthermore, this model concentrates on proactive flow rules, and the procedure for dealing with packets that do not match a rule are not described.

The controller and switch components call various functions to support their behaviour. Examples of controller functions are *SelectRoutes* which generates specific routings from the network topology, *CalcStats* and *CalcFlow* which takes switch counter information and calculates overall flows between hosts which can be then used by functions to update the routing array. Additional functionality within the SDN paradigm can be added by providing new functions rather than modifying the components, in most cases. It is also possible to add novel behaviour that SDN does not currently support such as probabilistic choice of rules to decide forwarding of packets in switches. With just these four component types and a separate network description, it is possible to create large network examples for analysis.

The host component (Fig. 3) allows for different traffic patterns specified as separate parallel components. An example of this is MapReduce which will be used in the experiments. Details of these have been elided for reasons of space. However, the basic idea is that certain traffic patterns will either be switched on or off as indicated by a Boolean attribute in the component. When a pattern is on, a packet corresponding to that traffic pattern can be sent. The host component repeatedly generates and sends packets, keeping count of the number. For measurement of packet latency, the time a packet is sent is included in the packet itself, together with the source, destination and protocol. In parallel, the host receives packets for which it is the destination and counts them. Both of these are done using the comm action together which a predicate that ensures that communication can only occur between connected network elements. This is discussed further in Sect. 5.2 below.

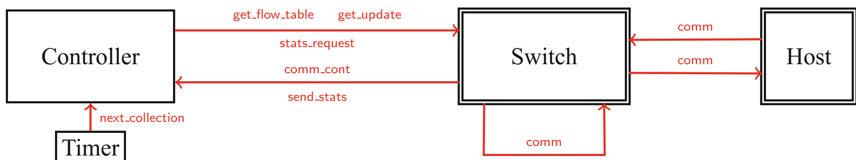


Fig. 2. Interaction of generic components

The switch component (Fig. 3) first receives its rule table from the controller and then processes incoming packets that are put into a queue when they arrive. For each packet, a matching rule is sought from the flow table where the flow table is an array of rule records which include source host identifier, destination host identifier, protocol, action to be done and counter to record how many times the rule is used. The action associated with the rule is applied: a packet can be dropped, sent to the controller, or forwarded. Parallel components wait for communication from the controller for statistics requests and rule updates, and set appropriate flags. Packet processing must be paused to update rules or send the flow table to the controller whenever an update is available or a request has been sent. Switch components use a single action comm to communicate with each other and hosts, and the significance of this will be discussed in Sect. 5.2.

The controller component uses the function *GenTop* to generate all possible routes between each pair of switches from the space description (see Sect. 5.1 for details of this description). From this collection of routes, a routing array is obtained using the function *SelectRoutes* to describe the route that a packet from one host to another will take. In SDN, exactly one route is chosen for a flow of packets from one host to another. In the model, the routes chosen can be balanced with respect to the number of hosts, so flows are evenly spread across core switches. They can also be unbalanced where a single core switch is used for all flows, or flows can be randomly assigned to different routes. The function *ConstructTables* is used to construct the flow tables for each switch which are then communicated to each switch, one by one. The controller component then interacts with the timer component to wait until it is time for the next collection of statistics, which is also done one-by-one from each switch. The counts in the flow tables of each switch are compared with the counts at the previous collection and traffic flows between hosts are obtained which can then be used to determine switch updates. As mentioned previously, this model considers proactive aspects of SDN rather than reactive, and hence the details of switch updates have not been included.

5.1 Space Syntax for Network Description

To specify a network description in a Carma SDN model requires the expression of the network topology in the space syntax which is independent of the generic components defined above, as well as traffic information for the network which is captured in two arrays for a specific traffic pattern. The first specifies the rate

Store of *Host* component:

host	host identifier
loc	location of switch that host is attached to
in	number of packets input
out	number of packets output
...	traffic pattern information for each traffic type

Behaviour of *Host* component:

$$Output \stackrel{\text{def}}{=} \sum_i \mathsf{gen_pkt_type}_i^{\star}[\bot]\langle\rangle\{out_pkt \leftarrow GenPktType_i(host, \mathsf{now}, \ldots)\}.Send$$

$$Send \stackrel{\text{def}}{=} \mathsf{comm}[ReceiverIsConn(\mathsf{my}.loc, loc)]\langle out_pkt\rangle\{out \leftarrow out + 1\}.Output$$

$$Input \stackrel{\text{def}}{=} \mathsf{comm}[SenderIsConn(\mathsf{my}.loc, loc) \wedge Dest(p) = host](p)\{in_pkt \leftarrow p\}.Log$$

$$Log \stackrel{\text{def}}{=} \mathsf{log_packet}^{\star}[\bot]\langle\rangle\{in \leftarrow in + 1\}.Input$$

$TP_i \stackrel{\text{def}}{=}$ traffic pattern behaviour for packet type$_i$

Initial state of *Host* component: $Output \mid Input \mid TP_1 \mid \ldots \mid TP_n$

Store of *Switch* component:

loc	switch identifier
flow_table	table of flow rules
q	queue of input packets
upd	if true then rule updates are available for installation
stats	if true then a stats request is pending

Behaviour of *Switch* component:

$$Start \stackrel{\text{def}}{=} \mathsf{get_flow_table}[SenderIsContr(loc)]\langle t\rangle\{flow_table \leftarrow t\}.Process$$

$$Process \stackrel{\text{def}}{=} [size(q) > 0]\mathsf{get_head}^{\star}[\bot]\langle\rangle\{pkt \leftarrow head(q), q \leftarrow tail(q)\}.Match$$

$$Match \stackrel{\text{def}}{=} \mathsf{find_rule}^{\star}[\bot]\langle\rangle\{rule \leftarrow FindRule(flow_table, pkt)\}.Count$$

$$Count \stackrel{\text{def}}{=} \mathsf{incr_count}^{\star}[\bot]\langle\rangle\{flow_table \leftarrow IncrCount(flow_table, rule)\}.Act$$

$$\begin{aligned} Act \stackrel{\text{def}}{=} {} & [Action(rule) = \mathrm{DROP}]\mathsf{skip}^{\star}[\bot]\langle\rangle.Update + \\ & [Action(rule) = \mathrm{CONT}]\mathsf{comm_cont}[ReceiverIsContr(loc)]\langle pkt\rangle.Update + \\ & [Action(rule) = \mathrm{FORW}]\mathsf{comm}[ReceiverIsConn(\mathsf{my}.loc, loc)]\langle pkt\rangle.Update \end{aligned}$$

$$\begin{aligned} Update \stackrel{\text{def}}{=} {} & [upd]\mathsf{rule_update}^{\star}[\bot]\langle\rangle\{flow_table \leftarrow RuleUpdate(rt, u), upd \rightarrow \bot\}.Stats + \\ & [\neg upd]\mathsf{skip}^{\star}[\bot]\langle\rangle.Stats \end{aligned}$$

$$\begin{aligned} Stats \stackrel{\text{def}}{=} {} & [stats]\mathsf{send_stats}^{\star}[ReceiverIsContr(loc)]\langle flow_table\rangle\{stats \rightarrow \bot\}.Process + \\ & [\neg stats]\mathsf{skip}^{\star}[\bot]\langle\rangle.Process \end{aligned}$$

$$Listen \stackrel{\text{def}}{=} \mathsf{comm}[SenderIsConn(\mathsf{my}.loc, loc)](p)\{q \leftarrow Append(q, p)\}.Listen$$

$$StatsReq \stackrel{\text{def}}{=} \mathsf{stats_request}[SenderIsContr(loc)]()\{stats \leftarrow \top\}.StatsReq$$

$$UpdRec \stackrel{\text{def}}{=} \mathsf{get_update}[SenderIsContr(loc)](u)\{upd \leftarrow \top, flow_table \leftarrow u\}.UpdRec$$

Initial state of *Switch* component: $Start \mid Listen \mid StatsReq \mid UpdRec$

Behaviour of *Timer* component:

$$NextCollect \stackrel{\text{def}}{=} \mathsf{next_collection}[loc = [Con]]\langle \mathsf{now}\rangle.NextCollect$$

Initial state of *Timer* component: $NextCollect$

Fig. 3. Host, switch and timer components

Store of *Controller* component:	
loc	controller identifier, always *Con*
top	topology, describes all routes between pairs of switches
routing	routing array, describes a single route between any pair of hosts
tables	array of flow tables, one entry for each switch
last_t	time of last statistics collection
sw_data	array of switch data, one entry for each switch
nwf_flow	total flow between each pair of hosts
sw_upd	array of switch updates, one for each switch

Behaviour of *Controller* component:

$$StartCon \stackrel{\text{def}}{=} \mathsf{gen_top}^{\star}[\bot]\langle\rangle\{top \leftarrow GenTop()\}.Routing$$

$$Routing \stackrel{\text{def}}{=} \mathsf{choose_routes}^{\star}[\bot]\langle\rangle.\{routing \leftarrow SelectRoutes(\top)\}.SwInit$$

$$SwInit \stackrel{\text{def}}{=} \mathsf{construct_tables}^{\star}[\bot]\langle\rangle\{tables \leftarrow ConstructTables(routing), i \leftarrow 0\}.SwInst$$

$$SwInst \stackrel{\text{def}}{=} [i < S]\mathsf{get_flow_table}[loc = [i]]\langle tables_i\rangle\{i \leftarrow i+1\}.SwInst + [i = S]\mathsf{skip}^{\star}[\bot]\langle\{i \leftarrow 0\}\rangle.Wait$$

$$Wait \stackrel{\text{def}}{=} \mathsf{next_collection}[t > last_t + collect_interval](t)\{last_t \leftarrow t\}.Request$$

$$Request \stackrel{\text{def}}{=} [i < S]\mathsf{stats_request}[loc = [i]]\langle\rangle.Collect + [i = S]\mathsf{calc_flow}^{\star}[\bot]\langle\rangle\{nwf_flow = CalcFlow(nwf_flow, sw_data), i \leftarrow 0\}.Analysis$$

$$Collect \stackrel{\text{def}}{=} \mathsf{send_stats}^{\star}[loc = [i]](d)\ \{sw_data_i \leftarrow CalcStats(d, \mathsf{now}, sw_data_i), i \leftarrow i+1\}.Request$$

Analysis $\stackrel{\text{def}}{=}$ (... decisions about changes to routing array based on *nwf_flow* resulting in new rules for switches stored in *sw_upd*...).*SwUpd*

$$SwUpd \stackrel{\text{def}}{=} [i < S]\mathsf{rule_update}[loc = [i]]\langle sw_upd_i\rangle.SwUpd + [i = S]\mathsf{skip}^{\star}[\bot]\langle\rangle\{i \leftarrow 0\}.Wait$$

Initial state of *Controller* component: *StartUpCon*

Fig. 4. Controller component

at which each host generates packets, and the second specifies the distribution of destinations for the packets, allowing for stochastic behaviour to be defined.

There is also second level of parametericity in CARMA SDN modelling. The size and shape of various topologies can be parametric and the space description can be defined to take this into account. For example, in the fat-tree topology, the parameter that specifies the size of the network is the number of ports in the type of switch used. For the experiments described later, the network topology description is parametric in this number, and increasing the parameter, increases the size of the network without time-consuming model updates as would be necessary with an emulator such as mininet.

The space syntax of CaSL, developed to model discrete space in collective adaptive systems, provides a mechanism to describe a directed graph, and this separates the network topology information from that of component behaviour. In the SDN model, each switch is assigned a unique location and each host is assigned the location of the switch to which it is attached (multi-homed hosts can not be modelled currently). Each edge in the network graph is described by `[a] -> [b] { port = p }`. This specifies that there is a network connection

```
const k = 6; // number of ports in each switch
space kPort_FatTree_Pod (){
  universe <int x>
  nodes { for i from 0 to k {[i];} }
  connections { for i from k/2 to k {
      for j from 0 to k/2 {
        [i] -> [j] { port=j };
        [j] -> [i] { port=i }; } } }
}
const Host_Switch = [: [3],[3],[3],[4],[4],[4],[5],[5],[5] :];
const Host_Port = [: 3,4,5,3,4,5,3,4,5 :];
```

Fig. 5. Space specification for one pod of Fig. 1 (partially parameterised)

from switch `a` to switch `b` and it is accessed through port `p` on switch `a`. Figure 5 illustrates how this language can be used to describe the left-most pod of Fig. 1 and the nine hosts it supports.

We number the six switches in the left-most pod with $0, \ldots, 6$ from left to right and top to bottom, and the six ports of a single switch are numbered in the same way. The nine hosts are numbered from 0 to 8. The bottom right switch of the pod is switch 5, with port 0 connected to switch 0, port 1 to switch 1 and port 2 to switch 2. Furthermore, port 3 is connected to host 6, port 4 to host 7 and port 5 to host 8. The ports 0, 1 and 2 of switches 0, 1 and 2 are not connected since we are considering the pod on its own for this example. Using this numbering of switches, ports and hosts, we can describe the network topology using the `space` keyword. Six locations are defined in the `nodes` section, one for each switch. In the `connections` section, links between switches are defined and labelled with port numbers. Only switch locations are defined, and two constant arrays are required to specify the location of each host in terms of the switch to which it is connected, and the port number of each host.

In the specification in Fig. 5, the actual space specification is parametric and hence solely dependent on the value of `k`. By contrast, the constant definitions are specific for `k = 6`. In the full SDN model, all of these are defined parametrically and hence a fat-tree topology of any size can be specified by changing the value of `k`.

The definition of the collective in Fig. 6 defines the location of each switch and host component using the @[...] notation. Each switch is located at its switch location, and *Host_Switch* is used to determine the location of each host. In the current model, hosts are not mobile, and hence their locations are fixed. However, it would be straightforward to have mobile hosts whose locations (represented by the switch to which they are currently attached) change over time. In this case, the array would be used to define the initial location of each host.

5.2 Space and Communication

All communication between switches and hosts use a single unicast action comm (we discuss communication between switches and controller below). The fact that a single action is used is exactly what allows for the definition of generic

components. If comm were replaced with actions that describe communication between specific components, it would be necessary to describe each network element as a separate component. In many process-algebra-style languages, the use of a single action would mean that all process with that action would be able to communicate with each other (assuming various forms of hiding are not used). Because CARMA implements attribute-based communication, predicates can be used to ensure that this free-for-all does not happen. The predicates used in the SDN model use the space description to determine which network elements are directly connected to each other. These are captured by the functions appearing in Fig. 3, specifically *ReceiverIsConn* and *SenderIsConn*. These functions take two arguments, my.*loc* and *loc*, which are the location of the sender and receiver respectively. In the case of communication between a switch and a host communication, it is necessary to check that the switch and host have the same location. In the case of communication between a switch and another switch, it is necessary to check that the receiver switch is in the post set of the sending switch with respect to the topology defined by the space description. This is checked with the syntax `loc in my.loc.post`. There are additional checks on port numbers which are defined on the connections.

By contrast, *ReceiverIsContr* and *SenderIsContr*, and the predicates $loc = [i]$ in the controller in Fig. 4 just check that the identifier is correct rather than for any connectivity. This provides for different levels of abstraction of networking in the model. To focus on network performance between hosts, we have chosen to model the movement of packets in the network between hosts at a fine-grained level (but not so fine-grained that we model varying packet sizes) whereas the communication between switches and controller is modelled more abstractly without packet-level details. This choice allows a focus on specific aspects of the model; in our case, the measurement of latency of actual traffic in the network. For the two scenarios we are considering, the secondary traffic between switches and controller is only for data collection and plays a negligible role and hence we can omit its detailed modelling.

5.3 The Collective and Environment

Figure 6 describes global constants and variables, together with the initial specification of the collective and the evolution functions of the environment. The number of ports in a switch is a global constant as are the total number of switches and host. Additional aspects of the space description are global constants. Globally, two variables are tracked. The number of packets that have been received, together with total amount of time taken by these packets in traversing the next allowing the creation of a measure to describe this with the following syntax.

```
measure average_latency
                    = global.pkt_time / global.pkt_count
```

As mentioned previously, the collective defines the individual copies of components with their locations. The timer component is not located. The evolution

Global constants:	k	number of ports in a switch
	S	total number of switches: $S = 5k^2/4$
	H	total number of hosts: $H = k^3/4$
	FatTree	description of network structure
	Host_Switch	array that allocates hosts to switches
	Host_Port	array that allocates hosts to ports
Global store:	*pkt_time*	sum of time taken by each packet
	pkt_count	total number of packets received
Collective:	*Timer*	single Timer component
	$Switch@[i]$	for $0 \leq i \leq S-1$
	$Controller@[Con]$	single Controller component
	$Host@[Host_Switch_j]$	for $0 \leq j \leq H-1$

Evolution rule functions:

$$\mu_p(\gamma_s, \gamma_r, \alpha) = 1$$

$$\mu_w(\gamma_s, \gamma_r, \alpha) = 1$$

$$\mu_r(\gamma_s, \alpha) = \begin{cases} \lambda_c & \alpha = \mathsf{comm} \\ \lambda_{ss} & \alpha = \mathsf{send_stats} \\ \lambda_{ru} & \alpha = \mathsf{rule_update} \\ \lambda_{cc} & \alpha = \mathsf{contr_comm}^\star \\ \ldots & \alpha = \mathsf{gen_pkt_type}_i^\star \\ \ldots & \text{other traffic pattern actions} \\ \mathit{fast} & \text{otherwise} \end{cases}$$

$$\mu_u(\gamma_s, \alpha) = \begin{cases} \{pkt_count \leftarrow pkt_count + 1 \\ \quad pkt_time \leftarrow pkt_time + (\mathsf{now} - \gamma_s(in_pkt.time_sent)\}, 0 \\ \qquad\qquad \alpha = \mathsf{log_packet}^\star \\ \{\}, 0 \qquad \text{otherwise} \end{cases}$$

Fig. 6. Global constants, global store, collective and environment functions

rule functions for probabilities and weights have a default value of 1 as these are not used in the SDN modelling. Most actions have the rate *fast* which means that they are essentially immediate. Other actions include communication, sending of statistics, updating of rules have constant rates, and as discussed previously traffic pattern rates may be based on information about senders and receivers.

The update function does nothing for all actions except $\mathsf{packet_log}^\star$. In the case of this action, when a packet is received by any host, the packet count is increased and the total time taken is increased appropriately, using information about when the packet was sent which is stored in the packet itself. This abstracts from the details relating to clock drift that can be involved in measuring packet latency in real networks. None of the actions introduce new components to the collective. In the SDN model described here, there is a fixed number of network elements which neither increase nor decrease during the model execution. In a model where hosts may appear and disappear, the update function would specify the action that would trigger the introduction of new host which would be

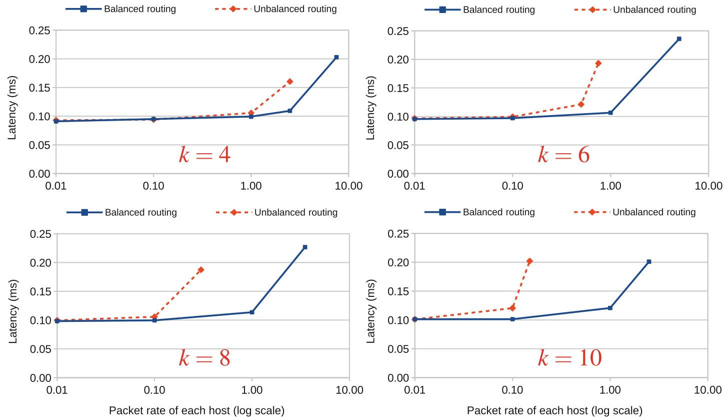


Fig. 7. Packet latency for balanced and unbalanced routing

instantiated with the relevant attribute values. Host components can disappear through the use of **kill**. It is also possible to add behaviour to capture the temporary impairment of all network elements.

6 Experiments and Results

We consider two experiments to illustrate the use of CARMA and MultiVeStA in considering the scalability of the fat-tree topology in the context of SDN and data centres [5]. These experiments report of the averages of measures and probabilities over multiple simulation runs performed by the CARMA software. We also mention the results of a security-focussed experiment that has been reported elsewhere [10]. These models assume the use of gigabit per second networks and have the communication and switch rates calibrated accordingly.

6.1 Scalability of Topology

This experiment considers two important aspects of fat-tree topology. In the first place, it compares the effect of multiple routes and how this allows for higher traffic loads to be viable. This is achieved by considered a balanced routing over the network and an unbalanced routing where all routes go through a single core switch of the network. Furthermore, we consider these for different number of ports, to assess whether the scaling is similar for different size networks. For this scenario, we assume equal traffic between each pair of hosts, and increase the rate of this traffic to the point that the switches start to become overloaded, and the packet latency increases so that the switches are unable to support the line speed of the network. These results are reported in Fig. 7. Data points are only included for experiments where latency reached steady state. The fat-tree topology scales moderately well when routing is balanced. For k of 6 or more,

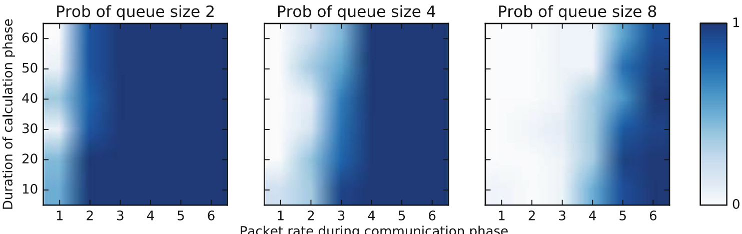


Fig. 8. Heat maps of queue size probabilities

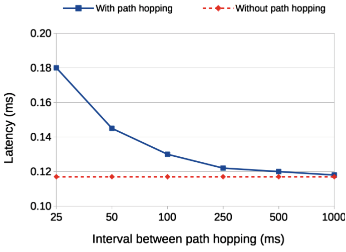
performance starts to drop off at a rate of 1.00. In the case of $k = 8$, there are 16256 flows at this rate, and for $k = 10$, there are 62250 flows at this rate (the table in Fig. 9 shows how the number of flows increase as k increases). In the case when routing is unbalanced to the extent that all flows go through a single core router, performance declines at lower rates as k increases. This emphasises the need for an SDN controller to utilise good load balancing algorithms to ensure good performance.

6.2 MapReduce Traffic Modelling

MapReduce describes a particular pattern of interaction between computers performing large computations [2]. A fixed number of hosts, say n, first perform a number of computations that can be done independently (the *map* phase). Once this phase is over, the results of the computation must be integrated via some computation and hence must be transferred to a single host (the *reduce* phase). This results in periods of limited traffic and then high traffic for the communication of results.

The SDN Carma model supports the modelling of this traffic pattern. When this pattern is activated, then there is an exponentially distributed duration in which there is limited communication between hosts, followed by a duration (drawn from a different exponential distribution) during which there are high levels of traffic from $n - 1$ of the hosts to the remaining host, and this alternating pattern repeats. We will refer to these as the *computation* phase and the *communication* phase, respectively.

In the previous experiment above that considered uniform traffic, it was assumed that the data centre network was being used by a single client (or alternatively that each host was shared by every client in the data centre) since there was traffic between all hosts. In the MapReduce experiment, a more constrained scenario is considered. First, for reasons of security, we assume that each host is used by a single client; second, we assume that each client has been allocated $2k^2/4$ hosts, and these hosts are connected to two pods. Thus, some of the traffic in the communication phase must traverse core switches to move between pods. For a fat-tree topology-based on k-port switches, this means that there are $k/2$ clients, each with a MapReduce traffic pattern (using the same exponential rates for periods of computation and communication).



k	4	6	8	10
1 tu	0.2s	1.3s	5.2s	19.4s
1000 tu	0.1h	0.4h	1.4h	5.4h
Hosts	16	54	128	250
Flows	240	2862	16256	62250
Packets	77.2K	132.5K	126.3K	248.5K
Packet rate	5.0	2.5	1.0	1.0
Components	38	101	210	377

Fig. 9. Packet latency for path hopping mitigation of the Sneak-Peek attack [10] (left). Execution times for different values of k (right).

We are interested in understanding the effect on maximum queue size of different rates for the MapReduce traffic pattern. We use a MultiVeStA query to determine the probabilities of a particular queue size for variations in the duration of the computation phase and variations in rate of traffic during the communication phase. We do not vary the duration of the communication phase. Figure 8 illustrates the probabilities obtained when considering a queue size of 2, 4 and 8. As can be seen, the rate of traffic during the communication phase has a greater impact on the queue size than the average duration of the gap (expresses as milliseconds) for the range of value considered. Since larger queue sizes (over 2) are associated with increased packet latency, these results demonstrate that decreased performance is likely to occur with packet rates over 4 for communication duration of up to 60 ms.

6.3 Trade-Offs in Network Security

We have also investigated security-performance trade-offs for the fat-tree topology [10] when considering mitigations of a covert channel attack call Sneak-Peek [41]. The mitigations involves changing route frequently (called *path hopping*) to disrupt an existing covert channel. Path hopping requires that new routes are sent to switches which causes delays in packet processing and increases latency. Realistic data for switch update delays have been used [37]. Experimentation (shown in Fig. 9) revealed that very frequent path hopping (every 25 ms) results in a 50% increase in latency. However, path hopping every second did not appear to increase the latency. Since the ability to create a covert channel with the Sneak-Peek attack requires shared network infrastructure, this model has been further investigated with MultiVeStA queries of the form "What is the probability that there will be a shared network component for a duration of x milliseconds?" thereby quantifying the risk of an attack [10].

7 Evaluation and Conclusion

We now evaluate the use of Carma and MultiVeStA and consider some metrics of relevance. We compare various metrics for the uniform traffic experiment

described in Sect. 6.1 and the figures in right-hand-side of Fig. 9 were obtained on a single core of a 2.66 Hz processor with 8 GB of memory available. Each time unit (tu) of simulation represents a millisecond of network time. The packet rate was chosen to be at a level where the network was fully loaded but not congested. The table includes the number of flows (between each pair of hosts), the packet rate and the number of packets transmitted in 1000tu. The last line is the number of components in the CARMA model. The time for the simulation of almost 20 s of network time in the 10-port case is moderately high; however, many independent simulations can be spread over different cores or computers, and hence averages over all simulations can be obtained in that time.

In terms of model construction, the fat-tree size can be increased by changing a single constant in the model code. Traffic patterns such as the MapReduce pattern can be specified parametrically as well. Hence it is possible to investigate regular topologies by only changing the model for the topology and relevant traffic patterns. Irregular network structures are likely to require more modifications of the mode code. The current model is around 1200 lines of CaSL code. Of this, about 25% of the code is components, collective, environment and measures; 10% is network and traffic specification; and 10% is parameters and data structure definition. The remainder are the function definitions required for the model, including the topology and routing determination, the predicates for constraining interaction between components, and calculations of network flows.

Further research includes comparison of model-building time and effort needed for the CARMA approach and the use of mininet, both for the initial model and for model changes; validation of small-scale CARMA models using mininet emulation, and experimentation with data and configurations from a real datacentre.

This research has demonstrated that CARMA is indeed suitable for modelling network and there is a good match between the discrete space syntax and the requirement for the description of network topology. Furthermore, this supports generic component definition, and leads to a model with a few generic components which aids comprehension. The richness of CARMA in terms of data structures and support for functions provides a step beyond what is usually possible in process-algebra-style modelling.

The comparison of the usability of our approach to others is hard to assess, partly because of different levels of abstraction. Certainly, our approach provides a level of abstraction that is novel in terms of assessing the performance of SDN networks. Furthermore, if we know that the generic components work correctly, then debugging can focus on the network specification. Hence, this level of abstraction allows for speed in developing, debugging and simulating models with a fast turn-around which is not possible with full-stack emulation or simulation methods, as shown by similar formal methods research [31]. Furthermore, very few formal methods take a quantitative approach which is required if performance is to be measured. The hybrid simulation approach using DEVS and TopoGen [8,30] is the closest to our research and works with network specification languages that could be used to obtain the CARMA network description. We

believe that our style of light-weight modelling style has a role to play, and the next phase of development is to embody our approach in software that conceals the details of the Carma from a user without experience of formal methods to provide a software tool which enables specification of network topologies and traffic patterns in a simple format with graphical elements where appropriate.

Acknowledgements. This work is supported by the EPSRC project Robustness-as-Evolvability, EP/L02277X/1 and the EPSRC Platform Grant EP/N014758/1. Thanks to David Aspinall, Wei Chen and Henry Clausen for their useful comments. The Carma models and experimental data can be obtained from groups.inf.ed.ac.uk/security/RasE/.

References

1. Abd Alrahman, Y., De Nicola, R., Loreti, M.: On the power of attribute-based communication. In: Albert, E., Lanese, I. (eds.) FORTE 2016. LNCS, vol. 9688, pp. 1–18. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39570-8_1
2. Al-Fares, M., Loukissas, A., Vahdat, A.: A scalable, commodity data center network architecture. In: Proceedings of the ACM SIGCOMM 2008, pp. 63–74 (2008)
3. Al-Shaer, E., Al-Haj, S.: FlowChecker: configuration analysis and verification of federated OpenFlow infrastructures. In: Proceedings of SafeConfig 2010, pp. 37–44 (2010)
4. Anderson, C., et al.: NetKAT: semantic foundations for networks. SIGPLAN Not. **49**, 113–126 (2014)
5. Bilal, K., et al.: Quantitative comparisons of the state-of-the-art data center architectures. Concurr. Comput.: Pract. Exp. **25**, 1771–1783 (2013)
6. Calheiros, R., Ranjan, R., Beloglazov, A., Rose, C.D., Buyya, R.: CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw. Pract. Exp. **41**, 23–50 (2011)
7. Canini, M., Venzano, D., Perešíni, P., Kostić, D., Rexford, J.: A NICE way to test OpenFlow applications. In: Proceedings of NSDI 2012 (2012)
8. Castro, R., Kofman, E.: An integrative approach for hybrid modeling, simulation and control of data networks based on the DEVS formalism. In: Modeling and Simulation of Computer Networks and Systems, pp. 505–551. Morgan Kaufmann (2015)
9. Chaves, L., Garcia, I., Madeira, E.: OFSwitch13: enhancing ns-3 with OpenFlow 1.3 support. In: Proceedings of WNS3 2016, pp. 33–40 (2016)
10. Chen, W., Lin, Y., Galpin, V., Nigam, V., Lee, M., Aspinall, D.: Formal analysis of Sneak-Peek: a data centre attack and its mitigations. In: IFIP SEC 2018 (2018, to appear)
11. Ciocchetta, F., Hillston, J.: Bio-PEPA for epidemiological models. Electron. Not. Theor. Comput. Sci. **261**, 43–69 (2010)
12. Dacier, M.C., Dietrich, S., Kargl, F., Knig, H.: Network attack detection and defense (Dagstuhl Seminar 16361). Dagstuhl Rep. **6**(9), 1–28 (2017)
13. De Nicola, R., Latella, D., Loreti, M., Massink, M.: A uniform definition of stochastic process calculi. ACM Comput. Surv. **46**, 5 (2013)
14. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: the SCEL language. ACM TAAS **9**, 7:1–7:29 (2014)

15. Farhadi, H., Lee, H., Nakao, A.: Software-defined networking: a survey. Comput. Netw. **81**, 79–95 (2015)
16. Fehnker, A., van Glabbeek, R., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 295–315. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_15
17. Fernandes, S.: Performance Evaluation for Network Services, Systems and Protocols. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-319-54521-9
18. Galpin, V.: Modelling network performance with a spatial stochastic process algebra. In: Proceedings of AINA 2009, pp. 41–49 (2009)
19. Galpin, V.: Modelling ambulance deployment with Carma. In: Lluch Lafuente, A., Proença, J. (eds.) COORDINATION 2016. LNCS, vol. 9686, pp. 121–137. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39519-7_8
20. Galpin, V., et al.: CaSL at work. QUANTICOL Deliverable D4.3 (2017). http://blog.inf.ed.ac.uk/quanticol/deliverables
21. Galpin, V., Zon, N., Wilsdorf, P., Gilmore, S.: Mesoscopic modelling of pedestrian movement using CARMA and its tools. ACM TOMACS **28**, 11:1–11:26 (2018)
22. Heller, B., Sherwood, R., McKeown, N.: The controller placement problem. In: Proceedings of HotSDN 2012, pp. 7–12 (2012)
23. Hillston, J.: A compositional approach to performance modelling. CUP (1996)
24. Hillston, J., Loreti, M.: Specification and analysis of open-ended systems with CARMA. In: Weyns, D., Michel, F. (eds.) E4MAS 2014. LNCS (LNAI), vol. 9068, pp. 95–116. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23850-0_7
25. Hillston, J., Loreti, M.: Carma eclipse plug-in: a tool supporting design and analysis of collective adaptive systems. In: Agha, G., Van Houdt, B. (eds.) QEST 2016. LNCS, vol. 9826, pp. 167–171. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43425-4_12
26. Ivey, J., Yang, H., Zhang, C., Riley, G.: Comparing a scalable SDN simulation framework built on ns-3 and DCE with existing SDN simulators and emulators. In: Proceedings of SIGSIM-PADS 2016, pp. 153–164 (2016)
27. Khurshid, A., Zou, X., Zhou, W., Caesar, M., Godfrey, P.: VeriFlow: verifying network-wide invariants in real time. In: Proceedings of NSDI 2013 (2013)
28. Kouzapas, D., Philippou, A.: A process calculus for dynamic networks. In: Bruni, R., Dingel, J. (eds.) FMOODS/FORTE 2011. LNCS, vol. 6722, pp. 213–227. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21461-5_14
29. Lantz, B., Heller, B., McKeown, N.: A network in a laptop: rapid prototyping for software-defined networks. In: Proceedings of Hotnets-IX, pp. 19:1–19:6 (2010)
30. Laurito, A., Bonaventura, M., Astigarraga, M., Castro, R.: TopoGen: a network topology generation architecture with application to automating simulations of software defined networks. In: Proceedings of WSC 2017, pp. 1049–1060 (2017)
31. Lemos, M., Dantas, Y., Fonseca, I., Nigam, V.: On the accuracy of formal verification of selective defenses for TDoS attacks. J. Log. Algebraic Methods Program. **94**, 45–67 (2018)
32. Loreti, M., Hillston, J.: Modelling and analysis of collective adaptive systems with CARMA and its tools. In: Bernardo, M., De Nicola, R., Hillston, J. (eds.) SFM 2016. LNCS, vol. 9700, pp. 83–119. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-34096-8_4
33. Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P., King, S.: Debugging the data plane with Anteater. In: Proceedings of SIGCOMM 2011 (2011)
34. Majumdar, R., Tetali, S., Wang, Z.: Kuai: a model checker for software-defined networks. In: Proceedings of FMCAD 2014, pp. 27:163–27:170 (2014)

35. Pascoal, T.A., Dantas, Y.G., Fonseca, I.E., Nigam, V.: Slow TCAM Exhaustion DDoS Attack. In: De Capitani di Vimercati, S., Martinelli, F. (eds.) SEC 2017. IAICT, vol. 502, pp. 17–31. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58469-0_2
36. Prakash, C., et al.: PGA: using graphs to express and automatically reconcile network policies. In: Proceedings of SIGCOMM 2015, pp. 29–42 (2015)
37. Rotsos, C., Sarrar, N., Uhlig, S., Sherwood, R., Moore, A.W.: OFLOPS: an open framework for OpenFlow switch evaluation. In: Taft, N., Ricciato, F. (eds.) PAM 2012. LNCS, vol. 7192, pp. 85–95. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28537-0_9
38. Sebastio, S., Vandin, A.: MultiVeStA: statistical model checking for discrete event simulators. In: Proceedings of ValueTools 2013, pp. 310–315 (2013)
39. Smolka, S., Kumar, P., Foster, N., Kozen, D., Silva, A.: Cantor meets Scott: semantic foundations for probabilistic networks. In: Proceedings of POPL 2017, pp. 557–571 (2017)
40. Son, S., Shin, S., Yegneswaran, V., Porras, P., Gu, G.: Model checking invariant security properties in OpenFlow. In: Proceedings of IEEE ICC 2013, pp. 1974–1979 (2013)
41. Tahir, R., et al.: Sneak-Peek: high speed covert channels in data center networks. In: Proceedings of IEEE INFOCOM 2016, pp. 1–9 (2016)
42. Zhou, W., Jin, D., Croft, J., Caesar, M., Godfrey, P.: Enforcing customizable consistency properties in software-defined networks. In: Proceedings of NSDI 2015 (2015)
43. Zoń, N., Gilmore, S., Hillston, J.: Rigorous graphical modelling of movement in collective adaptive systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 674–688. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_47

Resource-Aware Virtually Timed Ambients

Einar Broch Johnsen, Martin Steffen, Johanna Beate Stumpf(✉), and Lars Tveito

University of Oslo, Oslo, Norway
{einarj,msteffen,johanbst,larstvei}@ifi.uio.no

Abstract. Virtually timed ambients is a calculus of nested virtualization, which models timing and resource consumption for hierarchically structured virtual machines. This structure may change dynamically to support load-balancing, migration, and scaling. This paper introduces resource-awareness for virtually timed ambients, which enables processes to actively query the system about the resources necessary for a task and to reconfigure accordingly. Technically we extend virtually timed ambients with context-expressions using modal logic operators, give a formal semantics for the extension, and define bisimulation for resource-aware virtually timed systems. The paper also provides a proof of concept implementation in Maude and a case study involving dynamic auto scaling.

1 Introduction

In cloud-computing, *horizontal scaling* describes scaling by adding more machines into the given pool of resources. Cloud-service providers offer different kinds of scaling policies that allow their clients to monitor applications and automatically adjust capacity to maintain steady performance at low costs. For example, Amazon EC2 Auto Scaling [1] allows to dynamically and automatically scale the virtual capacity up or down according to conditions defined by the client. This paper provides a formalization to support dynamic auto scaling via resource-awareness for virtually timed ambients.

The calculus of *virtually timed ambients* [11] is a calculus of explicit resource provisioning, based on mobile ambients [3], and has been used to model nested virtualization in cloud systems. Virtualization technology enables the resources of an execution environment to be represented as a software layer, a so-called virtual machine. *Dynamic nested virtualization*, first introduced in [7], is a crucial technology to support cloud systems, as it enables virtual machines to migrate between different cloud providers [22]. It is also necessary to host virtual machines with operating systems which themselves support virtualization [2], such as Microsoft Windows 7 and Linux KVM. The time model used to realize the resource provisioning for virtually timed ambients is called *virtual time*. The time of a virtually timed ambient proceeds in the context of its parental ambient

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 194–213, 2018.
https://doi.org/10.1007/978-3-319-98938-9_12

and is relative to the parent's time progression, With nested levels of virtualization, virtual time becomes a *local* notion of time which depends on an ambient's position in the nesting structure. Virtually timed ambients are mobile, reflecting that virtual machines may migrate between host virtual machines. Observe that such migration affects the execution speed of processes in the migrating virtually timed ambient, as well as in the virtually timed ambient which is left, and in the virtually timed ambient which is entered.

Resource-awareness allows processes or programs to know about available resources and about resources necessary for a task, and react accordingly. For virtually timed ambients, resource awareness enables, e.g.,horizontal scaling, by adding more virtual machines to a server in the cloud. The notion of resource-aware virtually timed ambients is based on context-aware ambients (CCA) [21], which introduce context-guarded processes to enable context-awareness of mobile ambients. We enhance the given context expressions to cover the notions of timing and resources of virtually timed ambients and extend the theory of resource-aware virtually timed ambients by contextual bisimulation. We further provide a case study for modeling dynamic auto scaling on the cloud. Thus, we define a calculus to model explicit resource management in cloud computing.

Contributions. The main contributions of this paper are the following:

- we define and discuss a calculus of *resource-aware virtually timed ambients*;
- we define *weak timed context bisimulation* for resource-aware virtually timed ambients;
- we show the feasibility of virtually timed ambients as a modelling language for cloud computing with a *case study* of dynamic auto scaling on Amazon EC2 modelled in a prototype implementation of our calculus in the Maude rewriting system;
- all concepts are illustrated by *examples*.

To the best of our knowledge, this is the first implementation of resource awareness for mobile ambients in rewriting logic.

Paper Overview. We introduce resource-aware virtually timed ambients in Sect. 3. Section 4 discusses the implementation and contains the case study, exemplifying dynamic auto scaling on the cloud. We discuss related work and conclude in Sects. 5 and 6.

2 Virtually Timed Ambients

Virtually timed ambients [10,11] is a calculus of explicit resource provisioning, based on mobile ambients. Mobile ambients [3] are processes with a concept of location, arranged in a hierarchy which may change dynamically. *Virtually timed ambients* interpret these locations as places of deployment and extend mobile ambients with notions of virtual time and resource consumption. The timed behavior depends on the one hand on the *local* timed behavior, and on the other

Table 1. Syntax of virtually timed ambients, $x \in \mathbb{N}_0$.

n	name
`tick`	virtual time slice
Timed processes:	
$P, Q ::= \mathbf{0}$	inactive process
$\mid\ P \mid Q$	parallel composition
$\mid\ (\nu n)\, P$	restriction
$\mid\ !C.P$	replication
$\mid\ C.P$	prefixing
$\mid\ n[\textsc{Sched} \mid \texttt{tick}^x \mid P]$	virtually timed ambient
Timed capabilities:	
$C ::= \mathbf{in}\ n$	enter n and adjust the local scheduler there
$\mid\ \mathbf{out}\ n$	exit n and adjust the local scheduler on the outside
$\mid\ \mathbf{open}\ n$	open n and adjust own local scheduler
$\mid\ \mathbf{c}$	consume a resource

hand on the placement or deployment of the virtually timed ambient or the process in the hierarchical ambient structure. Virtually timed ambients combine timed processes and timed capabilities with the features of mobile ambients.

Definition 1 (Virtually timed ambients). *The syntax of virtually timed ambients is given by the grammar in Table 1.*

Timed processes differ from mobile ambients in that each virtually timed ambient contains, besides possibly further (virtually timed) subambients, a *local scheduler*. In the sequel, we omit the qualification "timed" or "virtually timed", when speaking about processes, capabilities, or ambients when the context of virtually timed ambients is clear. In the calculus, *virtually timed ambients* are represented by names and time slices are written as `tick`. The inactive process $\mathbf{0}$ does nothing. The parallel composition $P \mid Q$ allows both processes P and Q to proceed concurrently, where the binary operator $\mid$ is commutative and associative. The restriction operator $(\nu m)P$ creates a new and unique name with process P as its scope. Replication of processes is given as $!C.P$. A process P located in an virtually timed ambient named n is written $n[\textsc{Sched} \mid \texttt{tick}^x \mid P]$, where $\texttt{tick}^0 \equiv \mathbf{0}$. Ambients can be nested, and the nesting structure can change dynamically, this is specified by prefixing a process with a *capability* $C.P$. *Timed capabilities* extend the capabilities of mobile ambients by including a *resource consumption* capability $\mathbf{c}$ and by giving the *opening*, *exiting*, and *entering* capabilities of ambients a timed interpretation. These capabilities restructure the hierarchy of an ambient system, so the behavior of local schedulers and resource consumption changes, as these depend on the placement of the timed ambient in the hierarchy.

In a virtually timed ambient, the local scheduler triggers timed behavior and local resource consumption. Each time slice emitted by a local scheduler triggers

the scheduler of a subambient or is consumed by a process as a resource in a preemptive, yet *fair* way, which makes system behavior sensitive to co-located virtually timed ambients and resource consuming processes.

Definition 2 (Local and root schedulers). *Let the sets unserved and served contain the names of virtually timed ambients as well as processes (these are represented directly, lacking names). A* local scheduler *is denoted by*

$$\textsc{Sched}_{speed}\{in, out, rest, unserved, served\},$$

where speed $\in \mathbb{Q}$ *relates externally received to internally emitted time slices; in* $\in \mathbb{N}$ *records the number of received time slices; out* $\in \mathbb{N}$ *records the numbers of time slices than can be distributed for each incoming time slice, while rest* $\in \mathbb{N}$ *records additional distributable time slices depending on the speed; and unserved contains local ambients with a positive speed and processes which are intended to receive one time slice in this round of the scheduling, while served contains processes scheduled for the next round.*

Root schedulers, *represented as* $\textsc{Sched}^{\dagger}\{in, out, 0, unserved, served\}$*, are local schedulers which do not need an input to distribute time slices and therefore have no defined speed.*

The semantics of virtually timed ambients is given as a reduction system, similar to the semantics of mobile ambients. The rules for structural congruence $P \equiv Q$ are equivalent to those for mobile ambients (and therefore omitted here). The *reduction* relation $P \rightarrow Q$ for virtually timed ambients makes use of *observables,* also known as *barbs*. Barbs, originally introduced for the π-calculus [16], capture a notion of immediate observability. In the ambient calculus, these observations concern the presence of a top-level ambient whose name is not restricted. Let $\widetilde{m}$ describe a tuple of names, then the observability predicate $\downarrow_n$ or "barb" is defined as follows:

Definition 3 (Barbs, from [14]). *Process P strongly barbs on n, written* $P{\downarrow_n}$*, if* $P \equiv (\nu\widetilde{m})(n[P_1] \mid P_2)$*, where* $n \notin \{\widetilde{m}\}$*.*

A process that does not contain ν-binders is said to be *ν-binder free*. By moving the ν-binders to the outside and only considering the inside of their scope, we can observe the bound ambients inside the scope of the ν-binders.

Definition 4 (Timed top-level ambients). *For a process P, let* $P_{\downarrow}$ *denote the sets of all* timed top-level ambients*:* $P_{\downarrow} = \{n \mid P \equiv (\nu\widetilde{m})P' \wedge P' \text{ is } \nu\text{–binder free} \wedge P'{\downarrow_n} \wedge speed_n > 0\}$*.*

Timed Capabilities. The reduction rules for virtually timed ambients are given in Tables 2 and 3. The timed capabilities **in** n, **out** n, and **open** n enable virtually timed ambients to move in the hierarchical ambient structure. The local schedulers need to know about the current subambients, so their lists of subambients need to be adjusted when virtually timed ambients move. Observe that without adjusting the schedulers, the moving subambient would not receive

Table 2. Timed reduction rules for timed capabilities. Here, a blue backdrop marks the trigger of the reduction, red the changes in the schedulers, and green eventual constraints.

$$\dfrac{\begin{array}{l}\textsc{Sdl}_k = \textsc{Sched}_{speed_k}\{in_k, out_k, rest_k, U_k, S_k\},\ \ n \in U_k \cup S_k\\ \textsc{Sdl}_m = \textsc{Sched}_{speed_m}\{in_m, out_m, rest_m, U_m, S_m\}\\ \textsc{Sdl}_n = \textsc{Sched}_{speed_n}\{in_n, out_n, rest_n, U_n, S_n\}\\ \textsc{Sdl}'_k = \textsc{Sched}_{speed_k}\{in_k, out_k, rest_k, U_k \setminus \{n\}, S_k \setminus \{n\}\}\\ \textsc{Sdl}'_m = \textsc{Sched}_{speed_m}\{in_m, out_m, rest_m, U_m, S_m \cup \{n\}\},\ \text{if } speed_n > 0 \text{ else } \textsc{Sdl}_m\\ \textsc{Sdl}'_n = \textsc{Sched}_{speed_n}\{in_n, out_n, rest_n, U_n, S_n \cup P_\downarrow\}\end{array}}{\begin{array}{c}k[\textsc{Sdl}_k \mid n[\textsc{Sdl}_n \mid \textbf{in}\ m.P \mid Q] \mid m[\textsc{Sdl}_m \mid R] \mid U]\\ \rightharpoonup k[\textsc{Sdl}'_k \mid m[\textsc{Sdl}'_m \mid R \mid n[\textsc{Sdl}'_n \mid P \mid Q]] \mid U]\end{array}} \ (\text{TR-In})$$

$$\dfrac{\begin{array}{l}\textsc{Sdl}_k = \textsc{Sched}_{speed_k}\{in_k, out_k, rest_k, U_k, S_k\},\ \ n \in U_m \cup S_m\\ \textsc{Sdl}_m = \textsc{Sched}_{speed_m}\{in_m, out_m, rest_m, U_m, S_m\}\\ \textsc{Sdl}_n = \textsc{Sched}_{speed_n}\{in_n, out_n, rest_n, U_n, S_n\}\\ \textsc{Sdl}'_k = \textsc{Sched}_{speed_k}\{in_k, out_k, rest_k, U_k, S_k \cup \{n\}\},\ \text{if } speed_n > 0 \text{ else } \textsc{Sdl}_k\\ \textsc{Sdl}'_m = \textsc{Sched}_{speed_m}\{in_m, out_m, rest_m, U_m \setminus \{n\}, S_m \setminus \{n\}\}\\ \textsc{Sdl}'_n = \textsc{Sched}_{speed_n}\{in_n, out_n, rest_n, U_n, S_n \cup P_\downarrow\}\end{array}}{\begin{array}{c}k[\textsc{Sdl}_k \mid m[\textsc{Sdl}_m \mid n[\textsc{Sdl}_n \mid \textbf{out}\ m.P \mid Q] \mid R] \mid U]\\ \rightharpoonup k[\textsc{Sdl}'_k \mid n[\textsc{Sdl}'_n \mid P \mid Q] \mid m[\textsc{Sdl}'_m \mid R] \mid U]\end{array}} \ (\text{TR-Out})$$

$$\dfrac{\begin{array}{l}\textsc{Sdl}_k = \textsc{Sched}_{speed_k}\{in_k, out_k, rest_k, U_k, S_k\},\ \ n \in U_k \cup S_k\\ \textsc{Sdl}'_k = \textsc{Sched}_{speed_k}\{in_k, out_k, rest_k, U_k \setminus \{n\}, S_k \setminus \{n\} \cup P_\downarrow \cup R\downarrow\}\end{array}}{k[\textsc{Sdl}_k \mid \textbf{open}\ n.P \mid n[\textsc{Sdl}_n \mid R] \mid Q] \rightharpoonup k[\textsc{Sdl}'_k \mid P \mid R \mid Q]} \ (\text{TR-Open})$$

$$\dfrac{\begin{array}{l}\textsc{Sdl}_m = \textsc{Sched}_{speed_k}\{in_m, out_m, rest_m, U_m, S_m\},\ \ speed_m > 0\\ \textsc{Sdl}'_m = \textsc{Sched}_{speed_m}\{in_m, out_m, rest_m, U_m, S_m \cup \{\mathbf{c}\ .P\}\}\end{array}}{m[\textsc{Sdl}_m \mid \mathbf{c}\ .P \mid R] \rightharpoonup m[\textsc{Sdl}'_m \mid R]} \ (\text{TR-Resource})$$

time slices from the scheduler in its new surrounding ambient. In TR-In and TR-Out, the schedulers of the old and new surrounding ambient of the moving ambient are updated by removing and adding, respectively, the name of the moving ambient, if it has a speed greater zero. The scheduler of the moving subambient is also updated as it needs to contain the barbs of the process that was hidden behind the movement capability. In TR-Open, the scheduler of the opening ambient itself is updated by removing the name of the opened ambient

Table 3. Reduction rules for fair, preemptive distribution of virtual time and resources, where $b_y \in \mathbb{N}$. A blue backdrop marks the reduction trigger and red the changes.

$$\frac{\textsc{Sdl} = \textsc{Sched}_{speed}\{in, 0, 0, \emptyset, \emptyset\},\ \textsc{Sdl}' = \textsc{Sched}_{speed}\{in+1, 0, 0, \emptyset, \emptyset\},\ R \not\equiv \mathbf{c}\,.P \mid P'}{a[\,\texttt{tick} \mid \textsc{Sdl} \mid R] \twoheadrightarrow a[\textsc{Sdl}' \mid R]} \quad \text{(RR-Empty)}$$

$$\frac{\begin{array}{ll}\textsc{Sdl} = \textsc{Sched}_{speed}\{in, 0, 0, U, S\}, & U \cup S \neq \emptyset \\ \textsc{Sdl}' = \textsc{Sched}_{speed}\{in+1, x, z, U, S\}, & speed = x + \sum_{y=1}^{z} \frac{1}{b_y},\ b_y > 1\end{array}}{a[\,\texttt{tick} \mid \textsc{Sdl} \mid R] \twoheadrightarrow a[\textsc{Sdl}' \mid R]} \quad \text{(RR-Tick)}$$

$$\frac{\begin{array}{ll}\textsc{Sdl} = \textsc{Sched}_{speed}\{in, out, rest, \emptyset, S\}, & R \not\equiv \mathbf{c}\,.P \mid P' \\ \textsc{Sdl}' = \textsc{Sched}_{speed}\{in, out, rest, S, \emptyset\} & \end{array}}{a[\textsc{Sdl} \mid R] \twoheadrightarrow a[\textsc{Sdl}' \mid R]} \quad \text{(RR-NewRound)}$$

$$\frac{\begin{array}{l}out > 0,\ a_i \in U,\ a_i \equiv \mathbf{c}\,.P, \quad \textsc{Sdl} = \textsc{Sched}_{speed}\{in, out, rest, U, S\} \\ \textsc{Sdl}' = \textsc{Sched}_{speed}\{in, out-1, rest, U \setminus \{a_i\} \cup P_{\downarrow}, S\}\end{array}}{a[\textsc{Sdl} \mid R] \twoheadrightarrow a[\textsc{Sdl}' \mid R \mid P]} \quad \text{(RR-Tock}_{\text{1-consume}}\text{)}$$

$$\frac{\begin{array}{l}out > 0,\ a_i \in U, \quad R \equiv a_i[\textsc{Sdl}_{a_i} \mid P'] \mid P, \quad R' \equiv a_i[\textsc{Sdl}_{a_i} \mid \texttt{tick} \mid P'] \mid P \\ \textsc{Sdl} = \textsc{Sched}_{speed}\{in, out, rest, U, S\} \\ \textsc{Sdl}' = \textsc{Sched}_{speed}\{in, out-1, rest, U \setminus \{a_i\}, S \cup \{a_i\}\}\end{array}}{a[\textsc{Sdl} \mid R] \twoheadrightarrow a[\textsc{Sdl}' \mid R']} \quad \text{(RR-Tock}_{\text{1-ambient}}\text{)}$$

$$\frac{\begin{array}{l}rest > 0,\ in \bmod b_{rest} = 0,\ a_i \in U,\ a_i \equiv \mathbf{c}\,.P,\ speed = x + \sum_{y=1}^{z} \frac{1}{b_y},\ b_y > 1 \\ \textsc{Sdl} = \textsc{Sched}_{speed}\{in, out, rest, U, S\} \\ \textsc{Sdl}' = \textsc{Sched}_{speed}\{in, out, rest-1, U \setminus \{a_i\} \cup P_{\downarrow}, S\}\end{array}}{a[\textsc{Sdl} \mid R] \twoheadrightarrow a[\textsc{Sdl}' \mid R \mid P]} \quad \text{(RR-Tock}_{\text{2-consume}}\text{)}$$

$$\frac{\begin{array}{l}rest > 0,\ in \bmod b_{rest} = 0,\ a_i \in U, \quad speed = x + \sum_{y=1}^{z} \frac{1}{b_y},\ b_y > 1 \\ R \equiv a_i[\textsc{Sdl}_{a_i} \mid P'] \mid P, \quad R' \equiv a_i[\textsc{Sdl}_{a_i} \mid \texttt{tick} \mid P'] \mid P \\ \textsc{Sdl} = \textsc{Sched}_{speed}\{in, out, rest, U, S\} \\ \textsc{Sdl}' = \textsc{Sched}_{speed}\{in, out, rest-1, U \setminus \{a_i\}, S \cup \{a_i\}\}\end{array}}{a[\textsc{Sdl} \mid R] \twoheadrightarrow a[\textsc{Sdl}' \mid R']} \quad \text{(RR-Tock}_{\text{2-ambient}}\text{)}$$

$$\frac{\begin{array}{l}rest > 0,\ in \bmod b_{rest} \neq 0,\ speed = x + \sum_{y=1}^{z} \frac{1}{b_y},\ b_y > 1 \\ \textsc{Sdl} = \textsc{Sched}_{speed}\{in, out, rest, U, S\}, \quad \textsc{Sdl}' = \textsc{Sched}_{speed}\{in, out, rest-1, U, S\}\end{array}}{a[\textsc{Sdl} \mid R] \twoheadrightarrow a[\textsc{Sdl}' \mid R]} \quad \text{(RR-Tock}_{\text{2-no action}}\text{)}$$

$$\frac{\textsc{Sdl}^{\dagger} = \textsc{Sched}^{\dagger}\{in, 0, 0, U, S\},\ \textsc{Sdl}^{\dagger}_{*} = \textsc{Sched}^{\dagger}\{in+1, 1, 0, U, S\}}{\textsc{Sdl}^{\dagger} \twoheadrightarrow \textsc{Sdl}^{\dagger}_{*}} \quad \text{(RR-Root)}$$

and adding the barbs of the processes inside this ambient as well as the barbs of the process hidden behind the open capability. The scheduler of the opened ambient is deleted. In TR-Resource, the time consuming process moves into the scheduler, where it awaits the distribution of a time slice as resource before it can continue. This reduction can only happen in virtually timed ambients with speed greater zero, meaning ambients which actually emit resources.

The RR-Tick and RR-Tock rules in Table 3 distribute time slices via the local schedulers. We want to enable the schedulers to distribute time slices as soon as possible. The ratio of output time slices to input time slices is defined by the *speed* $\in \mathbb{Q}$ of the scheduler. For example, for a speed of $3/2$ the first incoming `tick` should trigger one outgoing time slice and the second input should trigger two, emitting in total three time slices for two inputs. Thus, in order to implement a simple *eager scheduling strategy*, we make use of the so-called *Egyptian fraction decomposition* to decide the number of time slices to be distributed by a local scheduler for each input time slice `tick`. For every rational number $q \in \mathbb{Q}$ it holds that $q = x + \sum_{y=1}^{z} \frac{1}{b_y}$ for $x, b_y \in \mathbb{N}$, which is solvable in polynomial time. A greedy algorithm (e.g. [6]) yields the desirable property that a time slice is distributed as soon as possible. From this decomposition, it follows that for each input time slice the local scheduler with speed q will distribute x time slices, plus one additional time slice for every b_y-th input. In RR-Tick, the local scheduler receives a time slice, which it registers in the counter *in*. At the same time *out* and *rest* initiate the distribution of time slices depending on the Egyptian fraction decomposition of the speed of the scheduler. These steps of the time slice distribution are shown in the RR-Tock rules, which allow transferring a new `tick` to a timed subambient or using the time slice as a resource for a consume capability, which is waiting in the scheduler. The RR-Tock_1 rules concern the number x of time slices that are given out for every input time slice, while the RR-Tock_2 rules only allow to give out a time slice if the input step is a multiple of one of the fraction denominators b_y. This amounts to a concrete implementation of a fair scheduler where progress is uniform over the queue of timed subambients and time consuming processes. Once all waiting subambients and processes inside the set *unserved* have been served one time slice and are moved to the set *served*, either the rule RR-NewRound ensures that the next round of time slice distribution can begin, or, if the queue is empty, the rule RR-Empty is applied. This scheduling strategy ensures fairness in the competition for resources between processes, without enforcing a particular order in each round of the scheduler. The root scheduler $\text{Sched}^\dagger$ reduces without time slices from surrounding ambients in RR-Root.

Example 1 (Virtually timed subambients, scheduling and resource consumption). The virtually timed ambient *cloud*, exemplifying a cloud server, emits one time slice for every time slice it receives, $\text{SDL}_{cloud} = \text{Sched}_1\{0, 0, 0, \emptyset, \emptyset\}$. It contains two `tick` and is entered by a virtually timed subambient *vm*.

$$\begin{aligned} &cloud[\text{Sched}_1\{0, 0, 0, \emptyset, \emptyset\} \mid \texttt{tick} \mid \texttt{tick}] \\ &\mid vm[\text{Sched}_{3/4}\{0, 0, 0, \emptyset, \emptyset\} \mid \textbf{in}\ cloud.\ \textbf{c}\ .P] \end{aligned}$$

The ambient *vm* exemplifies a virtual machine containing a resource consuming task, where $\text{SDL}_{vm} = \text{SCHED}_{3/4}\{0,0,0,\emptyset,\emptyset\}$. The Egyptian fraction decomposition of the speed yields $3/4 = 0 + 1/2 + 1/4$ meaning that there is no time slice given out for every incoming time slice, but one time slice for every second incoming time slice, and one for every fourth. The process reduces as follows:

$$
\begin{aligned}
&\rightarrowtriangle cloud[\text{SCHED}_1\{0,0,0,\emptyset,vm\} \mid \texttt{tick} \mid \texttt{tick} \\
&\quad \mid vm[\text{SCHED}_{3/4}\{0,0,0,\emptyset,\emptyset\} \mid \mathbf{c}\,.P]] && (\text{TR-IN}) \\
&\rightarrowtriangle cloud[\text{SCHED}_1\{0,0,0,vm,\emptyset\} \mid \texttt{tick} \mid \texttt{tick} \\
&\quad \mid vm[\text{SCHED}_{3/4}\{0,0,0,\emptyset,\emptyset\} \mid \mathbf{c}\,.P]] && (\text{RR-NEWROUND}) \\
&\rightarrowtriangle cloud[\text{SCHED}_1\{0,0,0,vm,\emptyset\} \mid \texttt{tick} \mid \texttt{tick} \\
&\quad \mid vm[\text{SCHED}_{3/4}\{0,0,0,\emptyset,\mathbf{c}\,.P\} \mid \mathbf{0}]] && (\text{TR-RESOURCE}) \\
&\rightarrowtriangle cloud[\text{SCHED}_1\{0,0,0,vm,\emptyset\} \mid \texttt{tick} \mid \texttt{tick} \\
&\quad \mid vm[\text{SCHED}_{3/4}\{0,0,0,\mathbf{c}\,.P,\emptyset\} \mid \mathbf{0}]] && (\text{RR-NEWROUND}).
\end{aligned}
$$

Here the ambient *vm* enters the ambient *cloud* and is registered in the scheduler. Furthermore, the resource consuming process in *vm* is registered. In the next steps the time slices move into the scheduler of the *cloud* ambient and are distributed further down in the hierarchy.

$$
\begin{aligned}
&\rightarrowtriangle cloud[\text{SCHED}_1\{1,1,0,vm,\emptyset\} \mid \texttt{tick} \\
&\quad \mid vm[\text{SCHED}_{3/4}\{0,0,0,\mathbf{c}\,.P,\emptyset\} \mid \mathbf{0}]] && (\text{RR-TICK}) \\
&\rightarrowtriangle cloud[\text{SCHED}_1\{1,0,0,\emptyset,vm\} \mid \texttt{tick} \\
&\quad \mid vm[\text{SCHED}_{3/4}\{0,0,0,\mathbf{c}\,.P,\emptyset\} \mid \texttt{tick}]] && (\text{RR-TOCK}_{\text{1-AMBIENT}}) \\
&\rightarrowtriangle cloud[\text{SCHED}_1\{2,0,0,vm,\emptyset\} \\
&\quad \mid vm[\text{SCHED}_{3/4}\{0,0,0,\mathbf{c}\,.P,\emptyset\} \mid \texttt{tick}]] && (\text{RR-NEWROUND}) \\
&\rightarrowtriangle cloud[\text{SCHED}_1\{2,1,0,vm,\emptyset\} \\
&\quad \mid vm[\text{SCHED}_{3/4}\{0,0,0,\mathbf{c}\,.P,\emptyset\} \mid \texttt{tick}]] && (\text{RR-TICK}) \\
&\rightarrowtriangle cloud[\text{SCHED}_1\{2,0,0,\emptyset,vm\} \\
&\quad \mid vm[\text{SCHED}_{3/4}\{0,0,0,\mathbf{c}\,.P,\emptyset\} \mid \texttt{tick} \mid \texttt{tick}]] && (\text{RR-TOCK}_{\text{1-AMBIENT}}) \\
&\rightarrowtriangle cloud[\text{SCHED}_1\{2,0,0,vm,\emptyset\} \\
&\quad \mid vm[\text{SCHED}_{3/4}\{0,0,0,\mathbf{c}\,.P,\emptyset\} \mid \texttt{tick} \mid \texttt{tick}]] && (\text{RR-NEWROUND}).
\end{aligned}
$$

Now the ambient *vm* can use the time signals to enable resource consumption.

$$
\begin{aligned}
&\twoheadrightarrow cloud[\textsc{Sched}_1\{2,0,0,vm,\emptyset\} \\
&\quad \mid vm[\textsc{Sched}_{3/4}\{1,0,1,\mathbf{c}\ .P,\emptyset\} \mid \texttt{tick}]] && (\text{RR-Tick}) \\
&\twoheadrightarrow cloud[\textsc{Sched}_1\{2,0,0,vm,\emptyset\} \\
&\quad \mid vm[\textsc{Sched}_{3/4}\{1,0,0,\mathbf{c}\ .P,\emptyset\} \mid \texttt{tick}]] && (\text{RR-Tock}_{2\text{-no action}}) \\
&\twoheadrightarrow cloud[\textsc{Sched}_1\{2,0,0,vm,\emptyset\} \\
&\quad \mid vm[\textsc{Sched}_{3/4}\{2,0,1,\mathbf{c}\ .P,\emptyset\} \mid \mathbf{0}]] && (\text{RR-Tick}) \\
&\twoheadrightarrow cloud[\textsc{Sched}_1\{2,0,0,vm,\emptyset\} \\
&\quad \mid vm[\textsc{Sched}_{3/4}\{2,0,0,P_{\downarrow},\emptyset\} \mid P]] && (\text{RR-Tock}_{2\text{-consume}})
\end{aligned}
$$

Note that as the calculus is non-deterministic, the reduction rules can be applied in arbitrary order, making several outcomes possible.

Table 4. Syntax of contexts and context expressions

Context:		**Context expressions:**	
$E ::= \mathbf{0}$	nil	$\kappa ::= \textsc{True}$	true
$\mid \odot$	hole	$\mid \odot$	hole
$\mid n[E]$	location	$\mid \neg\kappa$	negation
$\mid E \mid P$	parallel composition	$\mid \kappa_1 \mid \kappa_2$	parallel composition
		$\mid \kappa_1 \wedge \kappa_2$	conjunction
		$\mid n[\kappa]$	location
		$\mid \oplus\kappa$	spatial next modality
		$\mid \Diamond_{(speed,s)}\kappa$	somewhere modality
		$\mid \diamond_{x@n}\kappa$	sometime modality
		$\mid \exists x.\kappa$	existential quantification
		$\mid \mathbf{c}$	consumption

3 Resource-Aware Virtually Timed Ambients

We now consider *context-guarded actions* for the calculus of virtually timed ambients, building on properties of context aware ambients [21].

Definition 5 (Resource-aware virtually timed ambients). *The syntax of resource-aware virtually timed ambients is given by the grammar in Table 1 together with the process*

$$\kappa?P \ \textit{(context-guarded process)},$$

where κ is a context expression. The semantics of resource-aware virtually timed ambients is given by the reduction rules in Tables 2 and 3 and the rule

$$\frac{E \vDash \kappa}{E(\kappa?P) \twoheadrightarrow E(P)} \qquad (\text{TR-Context}).$$

A context-guarded process $\kappa?P$ has to fulfil a context requirement before it can be reduced, meaning that a guard is removed when it is satisfied by the environment. The context model is given in Table 4, where E denotes a context or environment and $\mathbf{0}$ is the empty context. Ambient names and processes are defined as in Table 1. The symbol $\odot$ is the hole context, showing the position of a process in the surrounding context. A ground context is defined to be a normal process with no holes. Multi-hole contexts are omitted.

Definition 6 (Context evaluation, from [21]). *Let E_1 and E_2 be contexts. The evaluation of context E_1 at context E_2, denoted $E_1(E_2)$, is the context obtained by replacing the hole in E_1 (if any) by E_2 as follows*

$$E_1(E_2) = \begin{cases} E_1 & \text{if } E_1 \text{ is a ground context,} \\ E_1\{\odot \leftarrow E_2\} & \text{otherwise,} \end{cases}$$

where $E_1\{\odot \leftarrow E_2\}$ is the substitution of E_2 for $\odot$ in E_1.

Context expressions are defined in Table 4. We enhance the context expressions for context-aware ambients from [21] with a *consumption* formula, stating the existence of consume capabilities in a process, as well as resource-aware *sometime* and *somewhere* modalities capturing the number of resources consumed in a certain ambient during the reduction, and the relative speed and number of siblings of the target ambient, respectively. To expose these numbers in reductions, we define a *labeled reduction relation.* While $\rightarrowtriangle$ refers to all reduction steps in virtually timed ambients, we denote by $\xrightarrow{\texttt{tick}}$ the steps of the (RR-Tick) rule, i.e., the internal reductions in the schedulers enabling timed reduction of processes. All other reduction steps are marked by $\xrightarrow{\tau}$.

Definition 7 (Tick-reduction). *$P \xrightarrow{\texttt{tick}} P'$ iff $P \mid \texttt{tick} \to P'$. We write $\xrightarrow{\texttt{tick}^x}$ if x time signals $\texttt{tick}$ are used; i.e., $P \xrightarrow{\texttt{tick}^x} P'$ iff $P \mid \texttt{tick} \mid \cdots \mid \texttt{tick} \to^* P'$, where the number of time signals $\texttt{tick}$ is x. The weak version of this reduction is defined as $P \xRightarrow{\texttt{tick}^x} P'$ iff $P(\xrightarrow{\tau}^* \xrightarrow{\texttt{tick}} \xrightarrow{\tau}^*)^x P'$, where $\xrightarrow{\tau}^*$ describes the application of an arbitrary number of τ-steps.*

The relation $\xRightarrow{\texttt{tick}^x}_n$ captures the number of resources used inside an ambient n inside a process.

Definition 8 (Tick-reduction inside an ambient). *$P \xRightarrow{\texttt{tick}^x}_n P'$ iff $P \rightarrowtriangle^* P'$ and there exists Q, Q' such that $P \downarrow^* n[Q], P' \downarrow^* n[Q']$ and $Q \xRightarrow{\texttt{tick}^x} Q'$.*

Lastly, we define *accumulated speed* [10] based on the eager distribution strategy for time slices. The accumulated speed $accum\{m\}_P \in \mathbb{Q}$ in a subambient m which is part of a process P, is the relative speed of the ambient m with respect to the speed of the parental ambient and the siblings of m.

Table 5. Satisfaction relation for context expressions

$E \models \text{TRUE}$		
$E \models \odot$	iff	$E = \odot$
$E \models \neg\kappa$	iff	$E \not\models \kappa$
$E \models \kappa_1 \mid \kappa_2$	iff	exist E_1, E_2, such that $E = E_1 \mid E_2$ and $E_1 \models \kappa_1$ and $E_2 \models \kappa_2$
$E \models \kappa_1 \wedge \kappa_2$	iff	$E \models \kappa_1$ and $E \models \kappa_2$
$E \models n[\kappa]$	iff	exist E', such that $E = n[E']$ and $E' \models \kappa$
$E \models \oplus\kappa$	iff	exist E', such that $E \downarrow\downarrow E'$ and $E' \models \kappa$
$E \models \Diamond_{(speed,s)}\kappa$	iff	exists E', E'', n s.t. $(E \equiv n[\text{SDL} \mid E'] \mid E'' \vee E \downarrow\downarrow^* n[\text{SDL} \mid E'])$ $\wedge E' \models \kappa \wedge accum\{n\}_E \geq speed \wedge \lvert U_{\text{SDL}} \cup S_{\text{SDL}} \rvert \leq s$
$E \models \diamond_{x@n}\kappa$	iff	exist E', such that $E \xrightarrow{\texttt{tick}^y}_n E'$, $y \leq x$ and $E' \models \kappa$
$E \models \exists x.\kappa$	iff	exist n, such that $E \models \kappa\{x \leftarrow n\}$
$E \models \mathbf{c}$	iff	exist E', E'', E''', such that $E \downarrow^* E'$ and $E' \equiv E''.\ \mathbf{c}\ .E'''$

Definition 9 (Accumulated speed). *Let $speed_k \in \mathbb{Q}$ and $children(k)$ denote the speed and number of children of a virtually timed ambient* k. *Let* m *be a timed subambient of a process P, the name* parent *denoting the direct parental ambient of* m, *and* C *the path of all parental ambients of* m *up to the level of P. The accumulated speed for preemptive scheduling in a subambient* m *up to the level of the process P is given by*

$$accum\{m\}_P = speed_m \cdot {}^1\!/_{children(parent)} \cdot speed_{parent}$$
$$= speed_m \cdot \prod_{k \in C} {}^1\!/_{children(k)} \cdot \prod_{k \in C} speed_k$$

Schedulers distribute time slices preemptively, as child processes get one time slice at a time in iterative rounds. Consequently, an ambient's accumulated speed is influenced by both the speed and the number of children n of the parental ambient. Thus, scheduling is not only *path sensitive* but also *sibling sensitive.*

The formal semantics for context expressions is given by the satisfaction relations in Table 5. The spatial reduction relation $\downarrow\downarrow$, which describes the option to go exactly one step deep into the nesting of ambients, is defined as follows.

Definition 10 (Spatial reduction). *$E \downarrow\downarrow E'$ iff there exist a name n and context E'' such that $E = (n[E'] \mid E'')$ and $\downarrow\downarrow^*$ is the reflexive and transitive closure.*

Thus, the *spatial next modality* $\oplus$ is satisfied if and only if the expression following it is satisfied after stepping one level down in the context. The *consumption* expression $\mathbf{c}$ is satisfied by any context which contains a consumption capability anywhere inside. A context E satisfies the *sometime modality* if and only if it can reduce to a context satisfying the formula, while using less than x resources in the ambient n in the reduction. Lastly, the *somewhere modality* is satisfied if and only if there exists a subcontext of E satisfying the formula and the relative speed in the sublocation containing the context is greater or equal the given *speed* and the sublocation has less or equal than s timed subambients.

We use the virtually timed system from Example 1 to show the meaning of some context expressions.

Example 2 (Context expressions and context-guards). It holds that

$$cloud[\textsc{Sdl}_{cloud} \mid \texttt{tick} \mid \texttt{tick} \mid vm[\textsc{Sdl}_{vm} \mid \mathbf{c}\,.\mathbf{0}]] \vDash \diamond_{2@vm}\neg\mathfrak{c}.$$

This means that two resources are used in the virtual timed ambient before the consume capability reduces. This reduction can be seen in Example 1. Further, it holds that

$$cloud[\textsc{Sdl}_{cloud} \mid \odot \mid \texttt{tick} \mid \texttt{tick} \mid vm[\textsc{Sdl}_{vm} \mid \mathbf{c}\,.P]] \vDash \oplus(vm[\textsc{True}] \mid \odot \mid \textsc{True})$$

or, omitting the hole,

$$cloud[\textsc{Sdl}_{cloud} \mid \texttt{tick} \mid \texttt{tick} \mid vm[\textsc{Sdl}_{vm} \mid \mathbf{c}\,.P]] \vDash \oplus(vm[\textsc{True}] \mid \textsc{True})$$

as there is an ambient named vm directly under the top level in the system. Using the context expression as context-guard we can define the following process

$$cloud[\textsc{Sdl}_{cloud} \mid \oplus vm[\textsc{True}]?\,\mathbf{open}\; vm \mid \texttt{tick} \mid \texttt{tick} \mid vm[\textsc{Sdl}_{vm} \mid \mathbf{c}\,.P]],$$

which aims to open the subambient vm if it is the only process on the top level in the system. As this is true after the ticks have been moved into the scheduler, the guard is removed and the process reduces to $cloud[\textsc{Sdl}_{cloud} \mid \mathbf{c}\,.P]$.

Weak Timed Context Bisimulation. We define *weak timed context bisimulation* for resource-aware ambients, which extends the definition of weak bisimulation for virtually timed ambients [11] by treating the context-guarded processes as τ actions in the timed labelled transition system and adding notions of context bisimulation [19,20] to the bisimulation relation.

The following definitions make use of the notion of *timed systems*, which are special processes without capabilities on the outermost level.

Definition 11 (Timed systems). *Timed systems are given as follows:*

$$\begin{aligned} M, N ::= {} & \mathbf{0} \\ \mid {} & M \mid N \\ \mid {} & (\nu n)M \\ \mid {} & n[P], \end{aligned}$$

where P is a timed process as given in Table 1.

The behavior of a timed system interacting with its environment is given as a transmission system with transition labels.

Definition 12 (Labels). *Let the set of labels Lab, with typical element α, be given as follows:*

$$\begin{aligned} \alpha \in Lab ::= {} & \tau \\ \mid {} & k.\texttt{enter_}n \mid k.\texttt{exit_}n \mid k.\overline{\texttt{enter}}\texttt{_}n \mid n.\texttt{open_}k \\ \mid {} & *.\texttt{exit_}n \mid *.\texttt{enter_}n \\ \mid {} & k.\texttt{tick} \end{aligned}$$

Table 6. Rules for timed labeled transition systems, where in (Co-Enter) given $\text{SDL}_k = \text{SCHED}_{speed_k}\{in, out, rest, unserved, served\}$ the updated scheduler is denoted by $\text{SDL}_k^* = \text{SCHED}_{speed_k}\{in, out, rest, unserved \cup \{n\}, served\}$ if $speed_n > 0$ as described in Table 2.

$(\nu\widetilde{m})(m[\text{SDL}\ \vert \mathbf{in}\ n.P \mid Q] \mid M), m \in \widetilde{m}$ $\xrightarrow{*.\mathtt{enter_}n} (\nu\widetilde{m})(n[m[(\text{SDL} \mid P) \mid Q] \mid \circ\ ] \mid M)$	(Enter Shh)
$(\nu\widetilde{m})(k[\text{SDL}\ \vert \mathbf{in}\ n.P \mid Q] \mid M), k \notin \widetilde{m}$ $\xrightarrow{k.\mathtt{enter_}n} (\nu\widetilde{m})(n[k[(\text{SDL} \mid P) \mid Q] \mid \circ\ ] \mid M)$	(Enter)
$(\nu\widetilde{m})(m[\text{SDL}\ \vert \mathbf{out}\ n.P \mid Q] \mid M), m \in \widetilde{m}$ $\xrightarrow{*.\mathtt{exit_}n} (\nu\widetilde{m})(m[(\text{SDL} \mid P) \mid Q] \mid n[M \mid \circ\ ])$	(Exit Shh)
$(\nu\widetilde{m})(k[\text{SDL}\ \vert \mathbf{out}\ n.P \mid Q] \mid M), k \notin \widetilde{m}$ $\xrightarrow{k.\mathtt{exit_}n} (\nu\widetilde{m})(k[(\text{SDL} \mid P) \mid Q] \mid n[M \mid \circ\ ])$	(Exit)
$(\nu\widetilde{m})(k[(\text{SDL}_k \mid P)] \mid M), k \notin \widetilde{m}$ $\xrightarrow{k.\overline{\mathtt{enter}}\mathtt{_}n} (\nu\widetilde{m})(k[\text{SDL}_k^* \mid n[\circ] \mid P] \mid M)$	(Co-Enter)
$(\nu\widetilde{m})(k[(\text{SDL} \mid P)] \mid M)$ $\xrightarrow{n.\mathtt{open_}k} n[\ \circ \mid (\nu\widetilde{m})(P \mid M)]$	(Open)
$(\nu\widetilde{m})(k[\text{SDL} \mid Q] \mid M), k \notin \widetilde{m}$ $\xrightarrow{k.\mathtt{tick}} (\nu\widetilde{m})(k[\text{SDL} \mid \mathtt{tick} \mid Q] \mid M)$	(Tick)

where k and n represent names of ambients. The label τ is called the internal label*, the rest are called* observable *labels. We refer to labels of the forms* $*.\mathtt{exit_}n$ *and* $*.\mathtt{enter_}n$ *as* anonymous *and other labels as* non-anonymous*, and let the* untimed *labels exclude the* $k.\mathtt{tick}$ *label.*

Note that the **c** capability does not represent an interaction with an environment but an internal action and is therefore not captured by a separate observable label apart from τ.

Definition 13 (Timed labeled transitions). *The observable steps $M \xrightarrow{\alpha} M'$ of the timed labeled transition semantics for timed systems is given by the rules of Table 6. For internal behavior, τ-steps are the result of reduction steps, i.e., $M \twoheadrightarrow M'$ implies $M \xrightarrow{\tau} M'$.*

The untimed labels, which record the system-environment interactions (i.e., ambient movements induced by capabilities), coincide with the labels from the untimed case of mobile ambients [14]. In rules Enter and Exit, an ambient k enters, respectively exits, from an ambient n provided by the environment. The rules Enter Shh and Exit Shh model the same behavior for ambients with private names. In rule Co-Enter, an ambient n, provided by the environment, enters an ambient k of the process. In rule Open, the environment provides an ambient n in which the ambient k of the process is opened. In rule Tick, the

transition $M \xrightarrow{k.\mathtt{tick}} M'$ expresses that the top-level ambient k of the system M receives one time slice tick from the root scheduler on the global level.

The post-configurations after the transitions contain the symbol $\circ$, which is used as placeholder variable. The labels, which capture interaction with the environment, carry partial information about the "data" exchanged with the environment. For example, label $k.\mathtt{enter_}n$ carries information about the identity k of the ambient being entered, which is contained in the system, as well as the identity of the entering ambient named n, which, before the step, is still part of the environment. If the enter-label conceptually indicates that some arbitrary ambient $n[R \mid \textsc{Sdl}]$ enters the system as an effect of executing the **in** n-capability, then the name n is mentioned as part of the label but its "body" $R \mid \textsc{Sdl}$ is not. We want to relate the actions of the two systems by a notion of bisimulation. Intuitively, if one system does a transition where $n[R \mid \textsc{Sdl}]$ enters, the second system must be able to exhibit the same transition, i.e., have the "same" ambient entering without breaking their (bi)simulation relationship. In principle, the second system can simulate the first doing a step where an ambient $n[R]$ enters, with the body $S \equiv R \mid \textsc{Sdl}$. To achieve that (without overburdening the labels by interpreting them up-to structural congruence $\equiv$), the definition uses the placeholder $\circ$ and requires preservation of the relationship for all *instantiations* of the placeholders for both systems by the same body (cf. Definition 14 below). The substitution of the placeholder by a pair consisting of a process and its local scheduler, is written as $P \bullet (\textsc{Sdl} \mid Q)$ and defined as expected.

The reduction semantics of a process can be encoded in the labelled transition system, because a reduction step can be seen as an interaction with an empty context. We are interested in bisimulations that abstract from τ-actions and use the notion of *weak actions*; let $\Longrightarrow$ denote the reflexive and transitive closure of $\xrightarrow{\tau}$, let $\overset{\alpha}{\Rightarrow}$ denote $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$, and let $\overset{\hat{\alpha}}{\Rightarrow}$ denote $\Longrightarrow$ if $\alpha = \tau$ and $\overset{\alpha}{\Rightarrow}$ otherwise.

Definition 14 (Weak timed context bisimulation). *A symmetric relation $\mathcal{R}$ over timed systems is a* weak timed context bisimulation *if $M \ \mathcal{R} \ N$ and $M \xrightarrow{\alpha} M', \alpha \in \{k.\mathtt{enter_}n, k.\mathtt{exit_}n, k.\overline{\mathtt{enter}}\mathtt{_}n, n.\mathtt{open_}k, *.\mathtt{exit_}n, *.\mathtt{enter_}n, k.\mathtt{tick}, \tau\}$ implies:*

1. *If α is a non-anonymous label, then $N \overset{\hat{\alpha}}{\Rightarrow} N'$ for some N', such that such that for all schedulers* Sched *and processes P it holds that $E[M' \bullet (\textsc{Sched} \mid P)] \ \mathcal{R} \ E[N' \bullet (\textsc{Sched} \mid P)]$, for each context E.*
2. *For anonymous labels:*
   (a) *If $\alpha = *.\mathtt{enter_}n$, then $N \mid n[\circ] \Longrightarrow N'$ for some N', such that for all schedulers* Sched *and processes P it holds that $E[M' \bullet (\textsc{Sched} \mid P)] \ \mathcal{R} \ E[N' \bullet (\textsc{Sched} \mid P)]$, for each context E.*
   (b) *If $\alpha = *.\mathtt{exit_}n$, then $n[\ \circ \mid N] \Longrightarrow N'$ for some N', such that for all schedulers* Sched *and processes P it holds that $E[M' \bullet (\textsc{Sched} \mid P)] \ \mathcal{R} \ E[N' \bullet (\textsc{Sched} \mid P)]$, for each context E.*

The preservation of bisimilarity by system contexts follows from this definition:

Theorem 1. *Weak timed context bisimilarity is preserved by system contexts.*

The given bisimulation relation is a congruence. Furthermore, the relation coincides with reduction barbed congruence, defined as the largest relation which is preserved by all constructs of the language, by the internal steps of the reduction semantics, and by so-called barbs, which are simple observables of terms.

Definition 15 (Reduction barbed congruence over timed systems). Reduction barbed congruence over timed systems *is the largest symmetrical relation over timed systems which is preserved by all system contexts, is reduction closed and barb preserving.*

We can show that the bisimulation relation coincides with reduction barbed congruence by following the proof given in [11].

Theorem 2. *Weak timed context bisimulation and reduction barbed congruence over resource-aware virtually timed systems coincide.*

4 Implementation and Case Study

We implement resource-aware virtually timed ambients in the rewriting logic system Maude [5,17]. Rewriting logic is a flexible semantic and logical framework which can be used to represent a wide range of systems with *low representational distance* [15]. Rewriting logic embeds *membership equational logic*, which lets a specification or program contain both equations and rewrite rules. When executing a Maude specification, rewrite steps are applied to normal forms in the equational logic. Both equations and rewrite rules may be *conditional*, meaning that specified conditions must hold for the rule or equation to apply.

The calculus of virtually timed ambient and a modal logic model checker for virtually timed ambients have been implemented in Maude [12]. The timed reduction rules (Tables 2 and 3) are represented as rewrite rules and modal logic formulas are built from operator declarations in Maude. We now extend this implementation with guarded processes and the corresponding reduction rule, and with replication and restricted names, thereby allowing non-unique names for ambients[1]. The syntax of resource-aware virtually timed ambients, given in Table 3, is represented by Maude terms, constructed from operators:

```
op zero : -> VTA [ctor] .
op _|_ : VTA VTA -> VTA [id: zero assoc comm] .
op _._ : Capability VTA -> VTA .
op _[_|_] : Name Scheduler VTA -> VTA .
op !_ : VTA -> VTA .
op !<_>_ : Names VTA -> VTA .
op _?_ : Formula VTA -> VTA [frozen (2)] .
```

[1] The full source code for the calculus and the case study is available at: https://github.com/larstvei/Check-VTA/tree/resource-aware.

The correlation between our formal definition and the Maude specification is easy to see. The operator `zero` represents the inactive process. Parallel composition has the algebraic properties of being associative, commutative and having `zero` as identity element. Concatenation is represented by a dot. Virtually timed ambients are represented by a name followed by brackets containing a scheduler and processes. Replication is represented by an exclamation mark and context-guarded processes by a question mark. The `frozen` attribute prevents subterms behind the guard from being rewritten before the guard has been resolved.

The prototype implementation currently covers a *negation free fragment* of the logic. Context expressions (defined in Table 4) are implemented explicitly as modal logic formulas, their duals have been implemented as necessary for the case study. We explain the implementation of the reduction rules by the rewrite rule for context-guards, corresponding to TR-Context. The guards express global properties, which make it necessary to capture the entire environment. This is achieved by wrapping the top-level ambient in brackets `op {_} : VTA -> VTA`, which syntactically distinguish the top-level ambient from subambients. Using these brackets, we can express that a rewrite rule may only be applied at the global level. Guards are resolved by invoking the given modal logic model checker for virtually timed ambients during execution:

```
crl [RemoveGuard] : { P } => { removeGuard(P, G) }
  if G, Gs := findGuards(P) /\
     removeGuardedProcess(P, G) |= G => true .
```

Here, `findGuards(P)` provides the set of all active guards found in the process, and some guard `G` is arbitrarily selected. Using the satisfaction relation `|=` the model checker is invoked on the top-level ambient, where the operation `removeGuardedProcess(P, G)` removes the guard together with the process behind it and thus yields the environment of the guarded process. If the environment satisfies the guard, the guard is removed by `removeGuard(P, G)`.

A Case Study of Dynamic Auto Scaling on Amazon EC2. In the following, we show how resource-aware virtually timed ambients can model dynamic auto scaling of Amazon EC2 instances, based on the Auto Scaling User Guide by Amazon Web Services [1]. An auto scaling group is a collection of EC2 instances, which are essentially virtual machines, illustrated in Fig. 1. The user can specify the minimum and maximum number of instances in an auto scaling group, and auto scaling ensures that the given group never goes below or above these bounds. By specifying scaling policies the user enables auto scaling to adjust the number of instances depending on the demand on the application.

We model a cloud server as a top-level ambient with a scheduler `sdl('asg)`, an auto scaling group `asg`, a garbage bin `garbage`, and a number of requests, demanding resources. A minimal example of scaling can be given by using two requests `request(2)`, each expecting two resources:

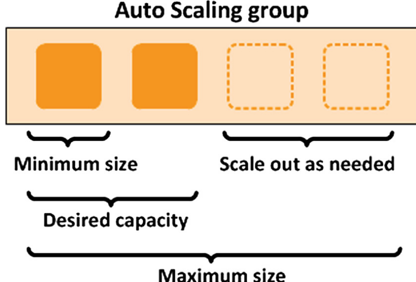


Fig. 1. Example of an auto scaling group as given in [1].

```
op example : -> VTA .
eq example =
    { 'cloud[sdl('asg) | asg | request(2) | request(2) | garbage] } .
```

The expressions `asg`, `garbage` and `request(2)` reduce to resource-aware virtually timed ambients, containing other ambients and processes. For example, `request(K)` is an ambient containing an empty scheduler `sdl`, movement capabilities and a number `K` of consume capabilities, representing load on the machine.

```
eq request(K) =
    'request[sdl | in('asg) . open('move) . zero | consumes(K)] .
```

The ambient `asg`, which models the auto scaling group, manages the virtual machines and dynamically scales depending on the load. A request may enter the `asg` where an idle virtual machine seizes it or, if no virtual machine is idle and the maximal number of virtual machines is not reached, the `asg` scales up and produces a new virtual machine to handle the request. Scaling up is achieved by means of replicated ambients with restricted names, representing new virtual machines, protected by a scaling guard which realizes the scaling policy:

```
eq scalingGuard(MIN, MAX) =
    (+) someone('asg[<> 0 MIN someone('isRegistry[True]) \/
                     someone('request[True]) /\
                     <> 0 MAX someone('isRegistry[True]) /\
                     no-one(NotConsume /\ (+) someone('isVM[True]))]) .
```

The guard checks the number of virtual machines, their load and the existence of a `request`. The formula `someone(F)` is introduced to capture a recurring pattern in the case study, namely the satisfaction of a formula by one ambient in the process. The dual is expressed by `no-one(F)`. The guard uses the somewhere modality and is satisfied if there are less then the minimal number of subambients in a `'registry` ambient (marked by the subambient `'isRegistry`) which contains a subambient for every active virtual machine. It is also satisfied if there

exists a request inside the auto scaling group, the maximal number of machines is not reached, and there is no idle virtual machine (marked by the property `NotConsume` and the subambient `'isVM`). Idle virtual machines move into the `garbage` ambient, if the number of virtual machines is not below the minimum. By running the `example` in Maude, we can see how the scaling process and the virtual machines react dynamically to the load on the auto scaling group.

```
Maude> frew example .
result VTA:
{'cloud[sched 0{0,0,0,'asg,none}
  | 'asg[sched 1{10,0,0,'vm1,none}
    | ...
    | !< 'vm > (open('scaling_lock) . scalingGuard ? scalingProcess)
    | 'vm1[...]]
  | 'garbage[sched 0{0,0,0,'vm0,none}
    | 'vm0[...] | ...]]}
```

Initially, the auto scaling group produces a virtual machine `'vm0[...]`, in accordance with the scaling policy which requires at least one running instance. The first request is handled by `'vm0[...]`, and a new virtual machine `'vm1[...]` is produced and handles the second request. Once `'vm0[...]` has resolved its request, it moves itself into the `garbage` ambient. The second virtual machine `'vm1[...]` is prevented from deleting itself, due to the scaling policy. The model autonomously creates virtual machines to deal with incoming requests and scales back down when the machines are not needed anymore.

5 Related Work

The calculus of virtually timed ambients, first introduced in [10], is based on mobile ambients [3]. Mobile ambients model both location mobility and nested locations, and capture processes executing at distributed locations in networks such as the Internet. Gordon proposed a simple formalism for virtualization (without notions of timing or resources) loosely based on mobile ambients in [8]. The calculus of virtually timed ambients [10,11] stays closer to the syntax of the original mobile ambient calculus, while at the same time including notions of time and explicit resource provisioning. Our notion of resource provisioning extends work on deployment models in ABS [9] to additionally cover nested virtualization and the capabilities of mobile ambients. Resource-awareness for virtually timed ambients draws on the Calculus of Context Aware Ambients [21] which introduces context-guarded processes to enable context-awareness of mobile ambients. The context expressions in this paper are adapted to cover the timing and resource aspects of virtually timed ambients.

Cardelli and Gordon defined a labeled transition system for their mobile ambients [4], but no bisimulation. A bisimulation relation for a fragment of mobile ambients, called mobile safe ambients, is defined in [13] and provides the

basis for later work. A labelled bisimulation for mobile ambients is defined by Merro and Nardelli [14], who prove that this bisimulation is equivalent to reduction barbed congruence and develop up-to-proof techniques. The weak timed bisimulation defined in [11] is a conservative extension of this approach, which is extended further in this paper using notions of context bisimulation developed in [19, 20].

In [12] we use the Maude [5] system to implement a model checker, exploiting the low representational distance which distinguishes Maude [15]. The reduction rules for mobile ambients as well as a type system have been implemented in Maude in [18]. In contrast, our implementation focuses on capturing the timed reduction rules of virtually timed ambients as well as the modal formulas to define guards and resource-awareness.

6 Concluding Remarks

Virtualization opens for new and interesting foundational models of computation by explicitly emphasizing deployment and resource management. The calculus of virtually timed ambients is a formal model of hierarchical locations of execution with explicit resource provisioning. Resource provisioning for virtually timed ambients is based on virtual time, a local notion of time reminiscent of time slices provisioned by an operating system to virtual machines in the context of nested virtualization. This paper introduces resource-awareness for virtually timed ambients, which enables horizontal scaling. We define weak timed context bisimulation for resource-aware virtually timed ambients as an extension of bisimulation for mobile ambients. We implement the calculus in the rewriting logic system Maude and illustrate its use by a case study of dynamic auto scaling. Future work aims to develop optimization strategies for resource-aware scaling as well as a notion of higher order resources.

References

1. Amazon Web Services. Auto Scaling User Guide. http://docs.aws.amazon.com/autoscaling/latest/userguide/as-dg.pdf. Accessed 21 June 2017
2. Ben-Yehuda, M., et al.: The Turtles project: design and implementation of nested virtualization. In: Proceedings 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010), pp. 423–436. USENIX Association (2010)
3. Cardelli, L., Gordon, A.D.: Mobile ambients. Theor. Comput. Sci. **240**(1), 177–213 (2000)
4. Cardelli, L., Gordon, A.D.: Equational properties of mobile ambients. Math. Struct. Comput. Sci. **13**(3), 371–408 (2003)
5. Clavel, M., et al. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
6. Fibonacci: Greedy Algorithm for Egyptian Fractions. https://en.wikipedia.org/wiki/Greedy_algorithm_for_Egyptian_fractions

7. Goldberg, R.P.: Survey of virtual machine research. IEEE Comput. **7**(6), 34–45 (1974)
8. Gordon, A.D.: V for virtual. Electron. Notes Theor. Comput. Sci. **162**, 177–181 (2006)
9. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Integrating deployment architectures and resource consumption in timed object-oriented models. J. Log. Algebr. Methods Program. **84**(1), 67–91 (2015)
10. Johnsen, E.B., Steffen, M., Stumpf, J.B.: A calculus of virtually timed ambients. In: James, P., Roggenbach, M. (eds.) WADT 2016. LNCS, vol. 10644, pp. 88–103. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72044-9_7
11. Johnsen, E.B., Steffen, M., Stumpf, J.B.: Virtually timed ambients: a calculus of nested virtualization. J. Log. Algebr. Methods Program. **94**, 109–127 (2018)
12. Johnsen, E.B., Steffen, M., Stumpf, J.B., Tveito, L.: Checking modal contracts for virtually timed ambients. In: Fischer, B., Uustalu, T. (eds.) ICTAC 2018. LNCS. Springer (2018, to appear)
13. Merro, M., Hennessy, M.: A bisimulation-based semantic theory of safe ambients. ACM Trans. Program. Lang. Syst. **28**(2), 290–330 (2006)
14. Merro, M., Nardelli, F.Z.: Behavioral theory for mobile ambients. J. ACM **52**(6), 961–1023 (2005)
15. Meseguer, J.: Twenty years of rewriting logic. J. Log. Algebr. Program. **81**(7–8), 721–781 (2012)
16. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 685–695. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55719-9_114
17. Ölveczky, P.C.: Designing Reliable Distributed Systems - A Formal Methods Approach Based on Executable Modeling in Maude. Undergraduate Topics in Computer Science. Springer, London (2018). https://doi.org/10.1007/978-1-4471-6687-0
18. Rosa-Velardo, F., Segura, C., Verdejo, A.: Typed mobile ambients in Maude. Electron. Notes Theor. Comput. Sci. **147**(1), 135–161 (2006). (In: Proceedings of the 6th International Workshop on Rule-Based Programming)
19. Sangiorgi, D.: Bisimulation for higher-order process calculi. Inf. Comput. **131**(2), 141–178 (1996)
20. Sangiorgi, D., Walker, D.: The Pi-Calculus: A Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
21. Siewe, F., Zedan, H., Cau, A.: The calculus of context-aware ambients. J. Comput. Syst. Sci. **77**(4), 597–620 (2011)
22. Williams, D., Jamjoom, H., Weatherspoon, H.: The Xen-Blanket: virtualize once, run everywhere. In: Proceedings of 7th European Conference on Computer Systems (EuroSys 2012), pp. 113–126. ACM (2012)

Stateful Behavioral Types for Active Objects

Eduard Kamburjan(✉) and Tzu-Chun Chen

Department of Computer Science, Technische Universität Darmstadt, Darmstadt, Germany
kamburjan@cs.tu-darmstadt.de, tc.chen@dsp.tu-darmstadt.de

Abstract. It is notoriously hard to correctly implement a multiparty protocol which involves asynchronous/concurrent interactions and constraints on states of multiple participants. To assist developers in implementing such protocols, we propose a novel specification language to specify interactions within multiple object-oriented actors and the side-effects on heap memory of those actors. A behavioral-type-based analysis is presented for type checking. Our specification language formalizes a protocol as a *global* type, which describes the procedure of asynchronous method calls, the usage of *futures*, and the heap side-effects with a first-order logic. To characterize runs of instances of types, we give a model-theoretic semantics for types and translate them into logical constraints over traces. We prove protocol adherence: If a program is well-typed w.r.t. a protocol, then every trace of the program adheres to the protocol, i.e., every trace is a model for the formula of the protocol's type.

1 Introduction

The combination of actors [25] with futures [4] in object-oriented languages (e.g., Scala [34] and ABS [28]), sometimes called *Active Objects* [12], is an active research area for system models and is frequently used in practice [37]. Processes of Active Objects communicate internally within an object via the object's heap memory. External communication works via asynchronous method calls with futures: constructs for synchronizing executions invoked by those calls. Encapsulated heap memory and explicit synchronization points make it easy to locally reason about Active Objects, but hard to specify and verify *global* protocols.

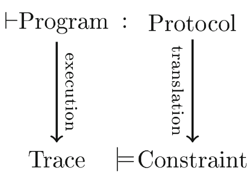
The main obstacle is to bridge the gap between local perspectives of single objects and global perspectives of the whole system. As Din and Owe [15] pointed out, it is non-trivial to precisely specify the communication within an object's heap memory from a global perspective [16]. Multiparty session types (short as MPST) [27], one important member of behavioral types [3,19], are established theories for typing *globally stateless* concurrent interactions (i.e., method calls) among multiple participants (i.e., objects) to ensure communication safety. Current works in MPST [6,38] have attempted to specify state in communication by using global values and assuming channels as the only communication concept.

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 214–235, 2018.
https://doi.org/10.1007/978-3-319-98938-9_13

Global values are not sufficient to specify the non-trivial interplay of processes when taking the heap memory inside of an object into account. Furthermore, channels are not able to fully represent the usage of futures, because futures, unlike channels, could expose some inner state of their object. Namely, it exposes that the computing process has terminated and the object was inactive before and after.

We integrate the stateful analysis and specification of traces of Din et al. [15] into MPST, where local verification of the endpoints compositionally guarantees the global specification of the whole system. Functional properties are specified as a part of the communication pattern. We ensure that from the perspective of each actor, its trace is not distinguishable from the global specification and that the whole system is deadlock free.

We specify passed data and modifications of heap memory with first-order logic (FOL) formulas and transform behavioral types into logical constraints on traces. Moreover, from the model-theoretic perspective, we define *protocol adherence* as the property that every generated trace of a well-typed($\vdash$) program is a model($\models$) for the translation of the type. The running example below illustrates the challenges for protocols in Active Objects.

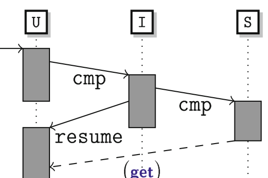


Consider a GUI `U`, a computation server `S`, and an interface server `I` such that `U`, without knowing `S`, wants to compute some data by sending it to `I` via a method call. After executing this call, `U` prepares for the next action by setting field `intern` to value `expect` and terminating its process to stay responsive. `I` delegates `U`'s task to `S` and remains responsive to other requests without waiting for `S`'s computation by invoking another method on `U` with future `x`, which will carry the computation result, back to `U`. The code and figure below implement this scenario:

```
1  object U{
2    TState intern = init;
3    Int resume(Fut⟨Int⟩ x){
4      if( this.intern!=expect } return −1;
5      Int r = x.get; return r; }
6    Unit start(Int j){
7      Fut⟨Unit⟩ f = I!cmp(j);
8      this.intern = expect; }}
```

```
9  object I{
10  Unit cmp(Int dat){
11    Fut⟨Int⟩ f = S!cmp(dat);
12    Fut⟨Int⟩ f’ = U!resume(f);}}
13
14 object S{ Int cmp(Int i){ ... }}
15
16 main { U!start(20); }
```

In the code, `!` denotes a non-blocking call, `I!cmp` calls method `cmp` of `I`, `U!start` calls `U.start`, `U!resume` calls method `resume` for continuation, and `S!cmp` starts the actual computation at `S`. The challenge for formal specifications is to express that (1) `I` is transparent to `U` and `S` such that `I` must pass the same data to `S` that it received from `U`, and `I` does not read the return value from `S`; and (2) `U` changes its heap to `expect` and reads the correct future.



Contributions. We propose (1) a specification language for actors' behaviors, that integrates FOL to specify heap memory, (2) model-theoretic semantics for protocol adherence, and (3) a static type system integrating a FOL validity calculus, which guarantees protocol adherence and deadlock freedom.

Roadmap. Section 2 provides an overview of our approach. Section 3 introduces a core language for Active Objects, Async, and its dynamic logic, Sect. 4 gives the types and operations on them and Sect. 5 gives the type system. Section 6 extends the concept to repetition. Section 7 concludes and discusses related work.

2 Scope, Challenges and an Overview of the Workflow

We aim to specify and verify *session-based* systems. A session-based system is a system which has a fixed, finite set of participating objects. Each object has an assigned *role* within the protocol of a session. Our analysis is fully static and is aimed at *system validation*: Ensuring that an existing system follows a certain specification.

We consider object-oriented actors, which use method calls, futures, and heap memory for communication. Every method call is asynchronous and starts a new process at the callee object. At each such call, the active *caller* obtains a *fresh* future identity, on which one may synchronize on the termination of the started process. An object may only switch its active process to another process if the currently active process terminates. The usage of futures provides programmers with the control of *when* synchronize – however, combining futures with object-oriented actors leads to the following complications:

Protocols with State. In an object-oriented setting, one must take the heap memory into account when reasoning about concurrent computations. For one, the heap memory influences the behavior of objects. For another, changes of the heap memory (among coordinated actors) are not only a by-effect of communication but often the *aim* of a protocol. Actors enforce strong encapsulation and restrict communication between object to asynchronous method calls and future reads – coordinated memory changes must be part of the specification.

Unexposed State. In the Active Object concurrency model, each process has exactly one future. Thus reading from a future is synchronizing with an unknown process *and depends on the state of the process's object.* To avoid deadlocks, futures cannot be analyzed in isolation—reading from a future must take the unexposed state of the object into account.

Mixed Communication Paradigms. Processes inside an object communicate through the heap memory. This kind of communication is hard to describe with data types, as it requires fine-grained specification of computation and has no explicit caller or callee. Thus, it is difficult to isolate the parts of the program which realize the communication protocol. Furthermore, method calls are asynchronous, while future reads are synchronous.

Two-Fold Endpoints. In the Active Object model, the callee endpoints of methods calls are *objects*, but the caller endpoints and the endpoints for future synchronization are *processes*. The interplay of multiple objects, which contain multiple processes, must be captured in the analysis by a two-fold notion of endpoints such that objects and processes are both endpoints.

In the following, we use the example from Sect. 1 to show how our approach works and addresses these issues.

Example 1: Specifying global types. Our specification language for side-effects is a FOL for specifying *local* memory instead of global values since (1) global values are not natural in an Active Object setting, and (2) a logic over memory locations (variables and fields) allows us to use a well-established theory of first order dynamic logic [22] to capture the semantics of methods. We formalize the scenario in Sect. 1 by the following global type in our specification language:

$$\mathbf{G} = \mathbf{main} \to \mathsf{U}\!:\!\mathtt{start}\langle \mathtt{U.state} \doteq \mathtt{expect}\rangle \ . \ \mathsf{U} \to \mathsf{I}\!:\!\mathtt{cmp}\langle \top, \top\rangle \ .$$
$$\mathsf{I} \xrightarrow{\mathtt{f}} \mathsf{S}\!:\!\mathtt{cmp}\langle \mathtt{i} \doteq \mathtt{dat}, \mathbf{result} > 0\rangle \ . \ \mathsf{I} \to \mathsf{U}\!:\!\mathtt{resume}\langle \mathtt{x} \doteq \mathtt{f}, \top\rangle \ . \ \mathsf{U} \uparrow \mathtt{x} \ . \ \mathsf{End}$$

We formally define the above syntax in Sect. 4 and only give the intuition here: $\top$ denotes true. $\mathsf{U} \to \mathsf{I} : \mathtt{cmp}$ denotes a message `cmp` from U to I, i.e., the call to a method `cmp`. Formula $\mathtt{U.state} \doteq \mathtt{expect}$ is the postcondition for the process *started* by this call at the callee object. If two formulas are provided, the first is the precondition describing the state of the caller and the second is the postcondition describing the state of the callee and the return value, which is denoted by keyword **result**. The annotation `f` denotes the memory location where the future of the denoted call is stored. Formula $\mathtt{i} \doteq \mathtt{dat}$ states that `dat`, the parameter of $\mathsf{S}.\mathtt{cmp}$, carries the same value as received by $\mathsf{I}.\mathtt{cmp}$ on parameter `i`, while formula $\mathtt{x} \doteq \mathtt{f}$ requires that parameter `x` of the call at method resume carries the future of the previous call to cmp. Finally, $\mathsf{U} \uparrow \mathtt{x}$ describes a read of `U` on the future stored in the location `x`. Note that we specify locations in formulas and avoid a situation where an endpoint must guarantee an obligation containing values that it cannot access. Other approaches (e.g., Bocchi et al. [6]) allow this situation and thus require additional analyses of history-sensitivity and temporal-satisfiability.

For the analysis, we adopt an approach similar to MPST: We project a global type on endpoints defined inside it, to automatically derive local specifications for all objects and methods. Additionally, formulas, which are used to specify conditions on the heap memory, are projected on the logical substructure of the callee, because the callee cannot access the caller's fields.

Two-Phase Analysis. The analysis requires that the protocol is encoded as a global type, which defines the order of method calls and future reads between objects, annotated with FO specifications of heap memory and passed data. Our analysis has two phases. In Phase 1, the global type is used to generate local types for all endpoints. In Phase 2, the endpoints are type checked against their

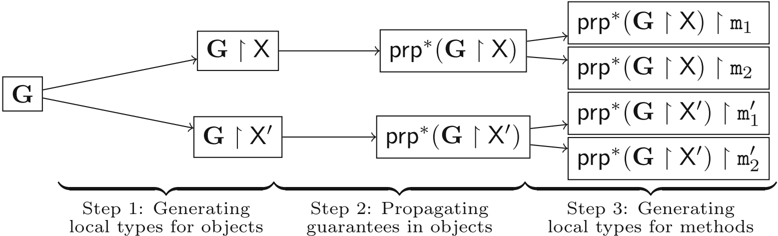


Fig. 1. Workflow for Phase 1: $\mathbf{G}$ is a global type and $\upharpoonright$ denotes projection on object X resp. method $\mathtt{m}$. Function prp^* is the function propagating guarantees.

local types and a causality graph is generated for checking for deadlocks. The workflow of Phase 1 is based on MPST's approach, but is adjusted to the Active Object concurrency model:

Phase 1. The workflow of Phase 1 is shown in Fig. 1.

- *Step 1*: The global type is projected onto the participating objects and generates *object* types. Such a type specifies the obligation of an object for running methods in a certain order, and for guaranteeing the FOL specifications of the object's state. During projection, the FO-specifications are projected onto the substructure of the object in question.
- *Step 2*: FO-specifications are propagated within an object type: as the order of method executions is specified by the specification, the postcondition of a method can be assumed as a *pre*condition for the next method.
- *Step 3*: An object type is projected on its methods, producing *method* types.

A global type encodes the following obligations (short as Obl.) for the implementation: (*Obl. a*) for each object, the observable events (calls and reads) are ordered as specified in the global type, (*Obl. b*) for each method, the observable events are ordered as specified in the local type derived from the global type and (*Obl. c*) the whole system does not deadlock, and adhere to the FO-specifications.

In the following, we demonstrate the workflow of Phase 1 for the global type in Example 1. We do not formally introduce the syntax at this point.

Step 1: Object Types. Projecting $\mathbf{G}$ from Example 1 on object $\mathtt{U}$ results in

$$?\mathtt{start}\langle\top\rangle.\mathsf{I}!\mathtt{cmp}\langle\top\rangle.\mathsf{Put}\ \mathtt{state} \doteq \mathtt{expect}.?\mathtt{resume}\langle\exists f.\ \mathtt{x} \doteq f\rangle.\mathsf{Read}\ \mathtt{x}.\mathsf{Put}\ \mathbf{result} > 0$$

Type $?\mathtt{start}\langle\top\rangle$ denotes a starting point for runtime execution. Type $\mathsf{I}!\mathtt{cmp}\langle\top\rangle$ denotes an invocation of method $\mathtt{cmp}$. Type $\mathsf{Put}\ \varphi$ specifies the termination of the currently active process in a state where φ holds. Position and postcondition of $\mathsf{Put}\ \mathtt{state} \doteq \mathtt{expect}$ are automatically derived. The position is just *before the next* method start and the postcondition is taken from the call in the global type. The analysis ensures that no method executes in-between. The precondition of $\mathtt{resume}$ is *weakened*, since field $\mathtt{f}$ is not visible to $\mathtt{U}$ and callee $\mathtt{U}$ cannot use all information from caller $\mathtt{I}$. Weakening ensures that all locations in φ are visible to $\mathtt{U}$. Type $\mathsf{Read}\ \mathtt{x}$ specifies a synchronization on the future stored in $\mathtt{x}$.

Step 2: Propagation. In the next step we propagate the postcondition of the last process to the precondition of the next process. No process is specified as active between `start` and `resume`, so the heap is not modified—thus, the postcondition of `start` still holds when `resume` starts. Adding $\texttt{state} \doteq \texttt{expect}$ to the precondition of `resume` strengthens the assumption for the type checking of `resume`. The propagation of conditions results in:

$$\mathsf{prp}^*(\mathbf{G} \upharpoonright \texttt{U}) = ?\texttt{start}\langle\top\rangle \ . \ \mathsf{I}!\texttt{cmp}\langle\top\rangle \ . \ \mathsf{Put}\ \texttt{state} = \texttt{expect} \ . \\ ?\texttt{resume}\langle\exists f.\ \texttt{x} \doteq f \wedge \texttt{state} \doteq \texttt{expect}\rangle \ . \ \mathsf{Read}\ \texttt{x} \ . \ \mathsf{Put}\ \mathbf{result} > 0$$

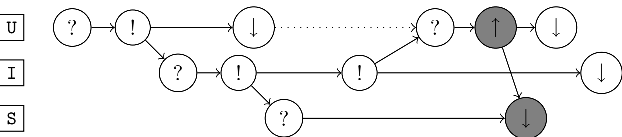
Step 3: Method Types. We generate a *method type* to specify a method in isolation. Projecting the object type in Step 2 on method `resume` generates:

$$\mathsf{prp}^*(\mathbf{G} \upharpoonright \texttt{U}) \upharpoonright \texttt{resume} = ?\texttt{resume}\langle\exists f.\ \texttt{x} \doteq f \wedge \texttt{state} \doteq \texttt{expect}\rangle.\mathsf{Read}\ \texttt{x}.\mathsf{Put}\ \mathbf{result} > 0$$

Method types share the syntax with object types. Projection from object types splits the object type at positions where one method ends and another one starts.

Phase 2. After generating method types, Phase 2 of the analysis checks the implementation of methods against their method types, and checks the formulas for validity. The type checking of method types guarantees the correct local order of events (*Obl. b*). State specifications are checked by integrating a validity calculus [15] into the type system. To guarantee (*Obl. a* and *c*), we require the following analyses:

Causality Graph. We generate a causality graph to ensure deadlock freedom (*Obl. c*): A deadlock free causality graph for Active Objects is cycle-free [17,24]. A causality graph is also used to ensure that methods of one object are executed in the order specified in the global type that the object obeys to (*Obl. a*).



The nodes are the local types from the projected object types. A solid edge connecting two nodes models that the statement for the first type directly causes the statement for the second type; for example, there are edges from a call to the corresponding receiving type. The graph is partially generated from **G**, and partially generated from the code: The edge connecting the gray nodes is added by a *Points-To* analysis, which maps a location of a future to the methods resolving this future. The termination of a method causes the start of the next (as the object cannot switch the active process otherwise), but does not select the next method itself. A dotted edge models such *indirect* causality: Indirect causality edges are considered when checking cycle-freedom check for deadlock freedom, but not for checking the method order.

Model-Theoretic Semantics. One of our contributions is the definition and verification of *protocol adherence* from a model-theoretic point of view: The property that a program follows a specified scenario (the protocol) if every generated trace is a model for the translation of the global type. We thus define protocol adherence through a *logical* characterization of global types and translate types into constraints over *traces*, which are sequences of configurations generated by the program.

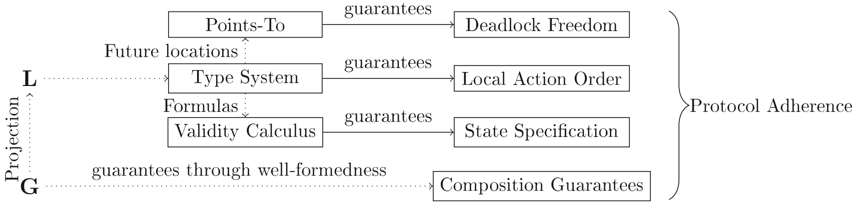


Fig. 2. Workflow for Phase 2 in our analysis.

This *declarative* approach for defining protocol adherence allows us to connect the FO properties embedded in the type to the execution of methods by using a dynamic logic: For a statement s the dynamic logic formula $\varphi \Rightarrow [\mathtt{s}]\psi$ expresses that the first-order formula ψ holds after executing s, if φ holds in the beginning. From the perspective of the trace logic, FOL describes a single configuration in the trace, while the modality $[\mathtt{s}]$ relates the configuration before executing s with the configuration after executing s. We use modalities during type checking.

3 Async, a Core Actor-Based Language Using Futures

We introduce Async, a simple Active Object language based on ABS [28]. Due to space limitations, we only present the basic constructs of Async below. For branching constructs we refer to [30]; repetition is introduced in Sect. 6. An Async-program consists of a main statement and a set of actors, which are objects that have fields and method but do not share state. Inside an object, processes do not interleave and the currently active process must terminate before another one is scheduled. Therefore, single methods can be considered sequential for analysis. We assume standard operations, literals and types for booleans, integers, lists and Object.

Definition 1 **(Async-Syntax).** *Let* e *denote expressions,* T *denote data types,* x *denote variable and field names,* X *denote object names, and* **Fut**<T> *denote future types.* $\bar{\cdot}$ *represents possibly empty lists and* $[\cdot]$ *represents optional elements.*

$$\mathsf{Prgm} ::= \overline{\mathsf{O}}\ \mathtt{main}\{\mathsf{X}!\mathtt{m}(\overline{\mathtt{e}})\} \quad \mathsf{O} ::= \mathtt{object}\ \mathsf{X}\ \{\overline{\mathsf{M}}\ \overline{\mathtt{T}\ \mathtt{x} = \mathtt{e}}\} \quad \mathsf{M} ::= \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}\ \mathtt{x}})\{\mathtt{s}; \textbf{return}\ \mathtt{e}\}$$
$$\mathtt{s} ::= \big[[\textbf{Fut}\texttt{<T>}]\ \mathtt{x} = \big]\mathsf{X}!\mathtt{m}(\overline{\mathtt{e}}) \mid [\mathtt{T}]\ \mathtt{x} = \mathtt{e} \mid [\mathtt{T}]\ \mathtt{x} = \mathtt{e}.\textbf{get} \mid \textbf{skip} \mid \mathtt{s}; \mathtt{s}$$

Objects communicate only by asynchronous method calls using futures. Upon a method call, a fresh future is generated on callee side and is passed to the caller. The callee writes the return value into the future upon termination of the corresponding process; anyone with the access to the future can read, but not write, into it. We only consider static sessions, in which all objects are created before the start of the system. Async is a standard imperative language with two additional statements: (1) $\mathtt{x} = \mathsf{X}!\mathtt{m}(\overline{\mathsf{e}})$ calls method $\mathtt{m}$ with parameters $\overline{\mathsf{e}}$ on object X. The generated future is stored in $\mathtt{x}$. The caller continues execution, while the callee is computing the call on $\mathtt{m}$ or scheduling $\mathtt{m}$ for later execution if another process is currently active. (2) e.**get** reads a value from the future stored in e. If the process computing this future has not terminated, the reading process blocks.

To define a small-step reduction relation over events for the semantics of Async, we first define an event as a process action with visible communication:

Definition 2 (Events). *Let f, f' range over futures. An* event*, denoted by* ev*, is defined by the following grammar:*

$$\mathsf{ev} ::= \mathsf{iEv}(\mathsf{X}, \mathsf{X}', f, \mathtt{m}, \overline{\mathsf{e}}) \mid \mathsf{iREv}(\mathsf{X}, f, \mathtt{m}) \mid \mathsf{fEv}(\mathsf{X}, f, \mathtt{m}, \mathsf{e}) \mid \mathsf{fREv}(\mathsf{X}, f, \mathsf{e}) \mid \mathsf{noEv}$$

An *invocation* $\mathsf{iEv}(\mathsf{X}, \mathsf{X}', f, \mathtt{m}, \overline{\mathsf{e}})$ models that X calls $\mathsf{X}'.\mathtt{m}$ using f and passes $\overline{\mathsf{e}}$ as parameters. An *invocation reaction* $\mathsf{iREv}(\mathsf{X}, f, \mathtt{m})$ models that X starts executing $\mathtt{m}$ to resolve f. A *resolving* $\mathsf{fEv}(\mathsf{X}, f, \mathtt{m}, \mathsf{e})$ models that X resolves f, which contains e at the moment, by finishing the execution of $\mathtt{m}$. A *fetch* $\mathsf{fREv}(\mathsf{X}, f, \mathsf{e})$ models that X reads value e from f. Finally, noEv models no visible communication.

A configuration is composed of *processes* and *objects*. A process has a unique future f, a store σ which maps variables to literals, and the name X of its object. An object has a unique name X, an active future f, and a store ρ which maps fields to literals.

Definition 3 (Runtime Syntax of Processes and Objects). *The following grammar defines runtime processes and objects as configurations* C*:*

$$\mathsf{C} ::= \mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{s}), \sigma) \mid \mathbf{prc}(\mathsf{X}, f, \mathsf{val}(\mathsf{e}), \sigma) \mid \mathbf{ob}(\mathsf{X}, f, \rho) \mid \mathsf{C}\ \mathsf{C}$$

A process either is executing a method $\mathtt{m}$ for a request carried by f at some object X, represented by $\mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{s}), \sigma)$, or has returned e, represented by $\mathbf{prc}(\mathsf{X}, f, \mathsf{val}(\mathsf{e}), \sigma)$. An object $\mathbf{ob}(\mathsf{X}, f, \rho)$ has its name X, the future of the active process f and the heap ρ. We write $\mathbf{ob}(\mathsf{X}, \bot, \rho)$ to indicate that X is inactive. Composition of configurations is commutative and associative, i.e., $\mathsf{C}\ \mathsf{C}' = \mathsf{C}'\ \mathsf{C}$ and $\mathsf{C}\ (\mathsf{C}'\ \mathsf{C}'') = (\mathsf{C}\ \mathsf{C}')\ \mathsf{C}''$. We denote the initial configuration of a program Prgm with $\mathbb{I}(\mathsf{Prgm})$. If all processes of a configuration C have terminated, the configuration also terminates. The body of method $\mathtt{m}$ is denoted by $M(\mathtt{m})$. We write $\widehat{M}(\mathtt{m}, \overline{\mathsf{e}})$ for the initial local store of a task executing $\mathtt{m}$ with parameters $\overline{\mathsf{e}}$.

We use *traces*, sequences of pairs of events and configurations, to capture the behavior of a program. We only consider terminating runs and define big-step semantics $\mathsf{Prgm} \Downarrow \mathbf{tr}$ for *finite* traces:

Definition 4 (Run and Big-Step Semantics). *A run from* C_1 *to* C_n *is a sequence of configurations* $\mathsf{C}_1, \dots, \mathsf{C}_n$ *with events* $\mathsf{ev}_1, \dots, \mathsf{ev}_{n-1}$ *such that:*

$$\mathsf{C}_1 \rightarrow_{\mathsf{ev}_1} \mathsf{C}_2 \rightarrow_{\mathsf{ev}_2} \dots \rightarrow_{\mathsf{ev}_{n-1}} \mathsf{C}_n$$

The trace $\mathbf{tr}$ *of a run is a sequence* $(\mathsf{ev}_1, \mathsf{C}_1), \dots, (\mathsf{ev}_m, \mathsf{C}_m)$ *where for every* $1 \leq j < m \leq n$ *there is a* C *such that* $\mathsf{C}_j \rightarrow_{\mathsf{ev}_j} \mathsf{C}$ *is in the run and* $\mathsf{ev}_j \neq \mathsf{noEv}$*. An* Async *program* Prgm *generates* $\mathbf{tr}$*, written* $\mathsf{Prgm} \Downarrow \mathbf{tr}$*, if there is a run from its initial configuration to a terminated configuration such that* $\mathbf{tr}$ *is the trace of this run.*

Figure 3 defines the reduction relation $\rightarrow_{\mathsf{ev}}$ for the semantics. $[\![\mathsf{e}]\!]_{\sigma,\rho}$ denotes the evaluation of an expression e under stores σ and ρ. Rule **(call)** executes a method call on the object stores in e by generating a fresh future f' and an invocation event. The new process is not set as active upon creation by **(call)**. By rule **(start)**, the object X must be inactive, when the process is started. An invocation reaction event is generated. Rule **(sync)** synchronizes on a future f' stored in e, by checking whether the configuration contains $\mathbf{prc}(\mathsf{X}', f', \mathsf{val}(\mathsf{e}'), \sigma')$, i.e. f' is resolved, and reads the return value e'. Rule **(end)** terminates a process. In all other rules, the ev parameter is noEv.

Dynamic Logic. A dynamic logic combines FO-formulas over the heap with symbolic executions [1,32] of statements. A symbolic execution uses symbolic values to describe a possible set of actual values. It does not reason about one execution of the statement, but describes a *set* of executions.

Example 2. Formula $\exists \mathtt{Int}\ a.\ \big(a > 0 \land \mathtt{i} > a\big) \rightarrow [\mathtt{j\ =\ i*2;}]\mathtt{j} > 0$ describes that if there is a number a bigger than 0 and smaller than the value stored in `i`, then after executing `j = i*2;`, variable `j` contains a positive value.

Based on ABSDL [14], we present Async Dynamic Logic (short as ADL), which extends first-order logic over program variables and heap memory with modalities that model the effect of statements. In this logic, method parameters are special variables and a modality is a formula $[\mathtt{s}]\varphi$ which holds in a configuration, say C, if φ holds in every configuration reached from C after executing `s`. We focus on the semantics of *modality-free* formulas, which have configurations as models; the semantics of modalities is a transition relation between configurations.

Definition 5 (Formulas φ). *We define the set of formulas* φ *and terms* t *by the following grammar, where* p *ranges over predicate symbols,* f *ranges over function symbols,* x *ranges over logical variables, and* v *ranges over logical and program variables. The set of formulas is denoted by* ADL*.*

$$\varphi ::= \mathbf{tt} \mid \neg\varphi \mid \varphi \lor \varphi \mid \mathsf{p}(\mathsf{t}\dots\mathsf{t}) \mid \mathsf{t} \geq \mathsf{t} \mid \mathsf{t} \doteq \mathsf{t} \mid \exists \mathtt{T}\ \mathsf{x}; \varphi \mid [\mathtt{s}]\varphi \qquad \mathsf{t} ::= \mathsf{v} \mid \mathsf{f}(\mathsf{t}\dots\mathsf{t})$$

Local program variables (i.e., `v`) are modeled as special function symbols. To model heap accesses, following Schmitt et al. [36], we use two function symbols store and select with (at least) the axiom $\mathsf{select}(\mathsf{store}(heap, \mathtt{f}, \mathtt{o}, value), \mathtt{f}, \mathtt{o}) =$

$$\textsf{(call)}\ \frac{\mathsf{C}\text{ does not contain } f' \quad [\![\mathsf{e}]\!]_{\sigma,\rho} = \mathsf{X}' \quad \mathsf{C} = \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathsf{C}' \quad \mathsf{ev} = \mathsf{iEv}(\mathsf{X}, \mathsf{X}', f, \mathtt{m}, [\![\overline{\mathsf{e}'}]\!]_{\sigma,\rho})}{\mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathsf{e}!\mathtt{m}'(\overline{\mathsf{e}'}); \mathtt{s}), \sigma)\ \mathsf{C} \rightarrow_{\mathsf{ev}} \mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{s}), \sigma)\ \mathbf{prc}(\mathsf{X}', f', \mathtt{m}'(M(\mathtt{m}')), \widehat{M}(\mathtt{m}, [\![\overline{\mathsf{e}'}]\!]_{\sigma,\rho}))\ \mathsf{C}}$$

$$\textsf{(start)}\ \frac{\mathsf{ev} = \mathsf{iREv}(\mathsf{X}, f, \mathtt{m})}{\mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{s}), \sigma)\ \mathbf{ob}(\mathsf{X}, \bot, \rho)\ \mathsf{C} \rightarrow_{\mathsf{ev}} \mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{s}), \sigma)\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathsf{C}}$$

$$\textsf{(sync)}\ \frac{\mathsf{C} = \mathbf{prc}(\mathsf{X}', f', \mathsf{val}(\mathsf{e}'), \sigma')\ \mathsf{C}' \qquad [\![\mathsf{e}]\!]_{\sigma,\rho} = f' \quad \mathsf{ev} = \mathsf{fREv}(\mathsf{X}, f', \mathsf{e}')}{\mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{x} = \mathsf{e}.\mathbf{get}; \mathtt{s}), \sigma)\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathsf{C} \rightarrow_{\mathsf{ev}} \mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{x} = \mathsf{e}'; \mathtt{s}), \sigma)\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathsf{C}}$$

$$\textsf{(end)}\ \frac{\mathsf{ev} = \mathsf{fEv}(\mathsf{X}, f, \mathtt{m}, \mathsf{e})}{\mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathbf{return}\ \mathsf{e}), \sigma)\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathsf{C} \rightarrow_{\mathsf{ev}} \mathbf{prc}(\mathsf{X}, f, \mathsf{val}([\![\mathsf{e}]\!]_{\sigma,\rho}), \sigma)\ \mathbf{ob}(\mathsf{X}, \bot, \rho)\ \mathsf{C}}$$

Fig. 3. The selected semantics rules. Full rules are provided in [30].

value where *heap* is a special local program variable modeling the heap explicitly. A special function symbol **result** is interpreted as the return value of a method, and a logical variable is *free* if it is not bound by any quantifier.

Definition 6. *A formula φ is* valid *if it evaluates to true in every configuration.*

Formulas are *global* or X-*formulas*. Global formulas refer to the heap of multiple objects, while X-formulas refer only to X. The latter contains only the function symbols for elements from X and the special function symbol **self** modeling the reference to X. For proving that an X-formula holds for a given state, if suffices to locally check the code of X. A validity calculus for ADL is presented in [15].

Definition 7. *Let φ be a formula. The weakened* X*-formula φ@X is obtained by replacing all function symbols in φ which are not exclusive to* X *(i.e., refer to the fields of other objects) by free variables and existentially quantifying over them.*

Example 3. Let fl be a field, X an object and i the parameter of some method in class X. Consider $\varphi = \mathsf{X.fl} > 0 \land \mathtt{i} > \mathsf{X.fl}$. The formula φ is an X-formula, as $\varphi = \varphi@\mathsf{X}$. The weakening for some object X′ is $\varphi@\mathsf{X}' = \exists \mathtt{Int}\ a.a > 0 \land \mathtt{i} > a$. $\varphi@\mathsf{X}'$ does not reason about X.fl, but still has the information of $\mathtt{i} > 1$.

4 Behavioral-Type-Based Stateful Specification

We define a specification language for global types to specify the behavior of the system. Following Sect. 3, we only represent the key constructs and leave branching to our technical report [30] and repetition to Sect. 6.

Definition 8 (Syntax of Global Types). Let φ, ψ range over modality-free ADLformulas and X_i range over object names. [·] denotes optional elements.

$$\mathbf{G} ::= \mathbf{main} \rightarrow \mathsf{X}{:}\mathtt{m}\langle\varphi\rangle.G \qquad G ::= \mathsf{X}_1 \xrightarrow{[\mathtt{x}]} \mathsf{X}_2{:}\mathtt{m}\langle\varphi, \psi\rangle.G \mid \mathsf{X}\uparrow\mathsf{e}.G \mid \mathsf{End}$$

The *calling type* $\mathsf{X}_1 \xrightarrow{\texttt{[x]}} \mathsf{X}_2 : \texttt{m}\langle\varphi,\psi\rangle$ specifies a method call from X_1 to $\texttt{m}$ at X_2. If $\texttt{x}$ is not omitted above the arrow, the future of this call must be stored in location $\texttt{x}$. The ADL-formula φ specifies (1) the call parameters passed to the callee and (2) the memory of X_1 at the moment of the call. Formula ψ is the postcondition of the callee process and specifies the state of X_2 and the return value once $\texttt{m}$ terminates. The exact point of termination is derived during projection. The initial method call $\mathbf{main} \to \mathsf{X}:\texttt{m}\langle\psi\rangle$ only specifies the postcondition of the process running $\mathsf{X}.\texttt{m}$. Type $\mathsf{X}\uparrow\mathsf{e}$ specifies a synchronization on the future, to which the expression e evaluates. Every synchronization must be specified. End specifies the end of communication.

$\mathbf{G}$ denotes a complete protocol with an initializing method call, while G denotes partial types. Even without fields in the formula, the implementation is referenced in the specification, as endpoints are object names. Object and method types share the same syntax. Together we call them *local types*. The grammar of local types is defined as follows:

Definition 9 (Syntax of Local Types). Let φ range over modality-free ADLformulas and X_i range over object names. $[\cdot]$ *denotes optional elements.*

$$\mathbf{L} ::= ?\texttt{m}\langle\varphi\rangle.L \qquad L ::= ?\texttt{m}\langle\varphi\rangle.L \mid \mathsf{X}!_{[\texttt{x}]}\texttt{m}\langle\varphi\rangle.L \mid \mathsf{Put}\ \varphi.L \mid \mathsf{Read}\ \mathsf{e}.L \mid \mathsf{skip}.L \mid \mathsf{End}$$

The type $?\texttt{m}\langle\varphi\rangle$ denotes the start of a process computing $\texttt{m}$ in a state where formula φ holds. Formula φ is the precondition of $\texttt{m}$ and describes the local state and method parameters of $\texttt{m}$. Type $\mathsf{Put}\ \varphi$ denotes the termination in a state where φ holds. Formula φ is a postcondition and describes the return value and the local store. Contrary to global types, a postcondition of a process is not annotated at the call, but at the point of termination because the point of termination is now explicit. Type $\mathsf{X}!_{[\texttt{x}]}\texttt{m}\langle\varphi\rangle$ corresponds to the caller side of $\mathsf{X}_1 \xrightarrow{\texttt{[x]}} \mathsf{X}_2 : \texttt{m}\langle\varphi,\psi\rangle$. Type $\mathsf{Read}\ \mathsf{e}$ models a read from e and skip denotes no communication. As for global types, End models the end of communication. In our examples, we omit End for brevity's sake. We use $\mathbf{L}$ for complete local types and L for partial local types.

Projection has three steps: (1) projection of global types on objects, (2) condition propagation, and (3) projection of object types on methods.

Projection on Objects. The projection on objects ensures that every object can access all locations occurring in its specification and adds $\mathsf{Put}\ \varphi$ at the correct position. This requires an additional parameter in the projection to keep track of which process is specified to be active and what its postcondition is.

To track the postcondition of the last active method of an object, we use a partial function $\mathsf{ac} : \mathsf{O} \rightharpoonup \mathsf{ADL}$ to map objects to formulas. If no method was active on X yet, ac is undefined, written $\mathsf{ac}(\mathsf{X}) = \bot$. The projection of G on an object X is denoted by $G \upharpoonright_{\mathsf{ac}} \mathsf{X}$. The selected projection rules for methods calls and termination are given in Fig. 4. We write $\mathsf{ac}_\bot$ for the function defined by $\forall \mathsf{X}.\ \mathsf{ac}(\mathsf{X}) = \bot$. For updates, we write $\mathsf{ac}[\mathsf{X} \mapsto \psi](\mathsf{X}') = \begin{cases} \psi & \text{if } \mathsf{X} = \mathsf{X}' \\ \mathsf{ac}(\mathsf{X}') & \text{otherwise} \end{cases}$.

$$\textbf{(1)}\ \mathsf{X}_1 \xrightarrow{\mathtt{x}} \mathsf{X}_2 : \mathtt{m}\langle\varphi,\psi\rangle.G \upharpoonright_{\mathsf{ac}} \mathsf{X}_1 = \mathsf{X}_2!_{\mathtt{x}}\mathtt{m}\langle\varphi\rangle.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2\mapsto\psi]} \mathsf{X}_1) \text{ if } \mathsf{ac}(\mathsf{X}_1) \neq \bot \wedge \varphi = \varphi@\mathsf{X}_1$$

$$\textbf{(2)}\ \mathsf{X}_1 \xrightarrow{\mathtt{x}} \mathsf{X}_2 : \mathtt{m}\langle\varphi,\psi\rangle.G \upharpoonright_{\mathsf{ac}} \mathsf{X}_2 = \begin{cases} ?\mathtt{m}\langle\varphi@\mathsf{X}_2\rangle.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2\mapsto\psi]} \mathsf{X}_1) & \text{if } \mathsf{ac}(\mathsf{X}_2) = \bot \\ \mathsf{Put}\ \mathsf{ac}(\mathsf{X}_2).?\mathtt{m}\langle\varphi@\mathsf{X}_2\rangle.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2\mapsto\psi]} \mathsf{X}_1) & \text{if } \mathsf{ac}(\mathsf{X}_2) \neq \bot \end{cases}$$

$$\textbf{(3)}\ \mathsf{X}_1 \xrightarrow{\mathtt{x}} \mathsf{X}_2 : \mathtt{m}\langle\varphi,\psi\rangle.G \upharpoonright_{\mathsf{ac}} \mathsf{X} = \mathsf{skip}.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2\mapsto\psi]} \mathsf{X}) \text{ if } \mathsf{X}_2 \neq \mathsf{X} \neq \mathsf{X}_1$$

$$\textbf{(4)}\ \mathbf{main} \rightarrow \mathsf{X}_2 : \mathtt{m}\langle\varphi\rangle.G \upharpoonright_{\mathsf{ac}_\bot} \mathsf{X}_1 = \begin{cases} ?\mathtt{m}\langle\varphi@\mathsf{X}_2\rangle.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2\mapsto\psi]} \mathsf{X}_1) & \text{if } \mathsf{X}_2 = \mathsf{X}_1 \\ \mathsf{skip}.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2\mapsto\psi]} \mathsf{X}_1) & \text{if } \mathsf{X}_2 \neq \mathsf{X}_1 \end{cases}$$

$$\textbf{(5)}\mathsf{End} \upharpoonright_{\mathsf{ac}} \mathsf{X} = \begin{cases} \mathsf{Put}\ \mathsf{ac}(\mathsf{X}).\mathsf{End} & \text{if } \mathsf{ac}(\mathsf{X}) \neq \bot \\ \mathsf{End} & \text{if } \mathsf{ac}(\mathsf{X}) = \bot \end{cases}$$

Fig. 4. The selected rules for projection on objects.

When projecting on caller X_1, a sending local type is generated by **(1)** if X_1 has an active process ($\mathsf{ac}(\mathsf{X}_1) \neq \bot$) and the precondition can be proven by the caller ($\varphi = \varphi@\mathsf{X}_1$). If the callee has an active process (i.e., the last active postcondition exists: $\mathsf{ac}(\mathsf{X}_2) \neq \bot$), then the termination type for the active process is added by **(2)** before the receiving type. If the callee is specified as being inactive (i.e., no process was running before and no postcondition is tracked $\mathsf{ac}(\mathsf{X}_2) = \bot$), then only the receiving type is added by **(2)**. When projecting on any other object, skip is added by **(3)**. In any case, ac is updated and maps the callee to a new postcondition. Rules **(4)** and **(5)** are straightforward. As usual, projection is undefined if no rule matches, and we omit $\mathsf{ac}_\bot$ and write just $\mathbf{G} \upharpoonright \mathsf{X}$.

Propagation. In our concurrency model the heap does not change if no process is active. All guarantees from the last active process still hold for the next process. By propagation, formulas are added from the postcondition of one method to the precondition of the next. Propagation moves formulas from where they *must hold* to all points where they still are *assumed to hold*. Propagation replaces a partial local type, if the partial type matches the given pattern.

Definition 10 (Propagation). *The propagation function* prp *is defined via term* rewriting *(denoted* $\rightsquigarrow$*) as follows.* prp^* *denotes the fixpoint of rewriting.*

$$\textbf{(1)}\ \mathsf{Put}\ \varphi.?\mathtt{m}\langle\psi\rangle \rightsquigarrow \mathsf{Put}\ \varphi.?\mathtt{m}\langle\psi \wedge \varphi@\mathsf{X}\rangle\ \textit{where}\,\mathsf{X}\ \textit{is the target object}$$

Projection on Methods. The projection on a method, denoted by $L \upharpoonright_{\mathtt{m}'} \mathtt{m}$, results in a *set* of method types. A method may have multiple method types, as long as the method types are *distinguishable*, which means that they have non-overlapping preconditions. Formally, two preconditions φ_1 and φ_2, are distinguishable if the formula $\neg(\varphi_1 \wedge \varphi_2)$ is valid. In the case of overlapping preconditions, multiple preconditions can hold at the same time and it is not guaranteed that the correct type will be realized.

The rules for projection on a method are straightforward and we refer to Sect. 2 for an example and to our technical report [30] for full definitions.

Definition 11 (Well-Formedness). *A global type* **G** *is* well-formed*, if the projections on all methods are defined and all types of a method are distinguishable.*

Semantics of Types as Constraints on Traces. To formalize the behavioral types of the previous section, we transform them into first-order constraints over traces.

We define $\mathbb{C}$ as a function transforming global types to constraints on traces. Recall that we have defined C for configurations and ev for events. The primitive $\mathsf{C}(i)$ references the ith configuration and $\mathsf{ev}(i)$ references the ith event in a trace. We use events and formulas as colors and thus include futures, method names, literals and object names in the domain. Constraints refer to ADL formulas φ with $\mathsf{C}(i) \models \varphi$, meaning that in the ith configuration, φ holds.

To restrict a constraint to a subtrace, we use *relativization* [23], a *syntactic* restriction of constraint γ to a substructure described by another constraint χ.

Definition 12. *Let $\chi(x)$ be a constraint with a free variable x of data type $\mathtt{T}$ and γ another constraint. The relativization of γ with $\chi(x)$, written $\gamma[x \in \mathtt{T}/\chi]$, replaces all subconstraints of the form $\exists y \in \mathtt{T}.\gamma'$ in γ by $\exists y \in \mathtt{T}.\chi(y) \wedge \gamma'$.*

The main rules for translating **G** into a constraint $\mathbb{C}(\mathbf{G})$ are defined as follows.

Definition 13 (Semantics of Global Types). *Predicate $\mathsf{res}(i)$ holds if $\mathsf{ev}(i)$ is a resolving event and $\mathsf{A}(i, \mathsf{X})$ holds if X is active in $\mathsf{C}(i)$.*

$$\textbf{(1)}\ \mathbb{C}(\mathbf{main} \rightarrow \mathsf{X}_2 : \mathtt{m}\langle\psi\rangle.G) \;=\; \exists j,k.\ \exists f.\ \exists \mathsf{e}'.\ \mathsf{ev}(j) \doteq \mathsf{iREv}(\mathsf{X}_2, f, \mathtt{m}) \wedge \mathsf{C}(j) \models \varphi @ \mathsf{X}_2 \wedge \\ \mathsf{ev}(k) \doteq \mathsf{fEv}(\mathsf{X}_2, f, \mathsf{e}') \wedge \mathsf{C}(k) \models \psi \wedge \forall l. l \neq j \wedge l \neq k \Rightarrow \mathsf{res}(l) \wedge \mathbb{C}(G)$$

$$\textbf{(2)}\ \mathbb{C}(\mathsf{X}_1 \xrightarrow{\mathtt{x}} \mathsf{X}_2 : \mathtt{m}\langle\varphi, \psi\rangle) \;=\; \exists i,j,k.\ \exists f.\ \exists \mathsf{e}, \mathsf{e}'. \\ \mathsf{ev}(i) \doteq \mathsf{iEv}(\mathsf{X}_1, \mathsf{X}_2, f, \mathtt{m}, \mathsf{e}) \wedge \mathsf{C}(i) \models \varphi \wedge \mathsf{ev}(j) \doteq \mathsf{iREv}(\mathsf{X}_2, f, \mathtt{m}) \wedge \mathsf{C}(j) \models \varphi @ \mathsf{X}_2 \wedge \\ \mathsf{ev}(k) \doteq \mathsf{fEv}(\mathsf{X}_2, f, \mathsf{e}') \wedge \mathsf{C}(k) \models \psi \wedge \mathsf{C}(i) \models (\mathsf{X}_1.\mathtt{x} \doteq f) \wedge \forall l.\ l \neq i \wedge l \neq j \wedge l \neq k \Rightarrow \mathsf{res}(l)$$

$$\textbf{(3)}\ \mathbb{C}(G_1.G_2) = \bigwedge_{\mathsf{X}} \left(\exists i \in \mathbb{N}.\ \mathbb{C}(G_1)[j \in \mathbb{N}/\mathsf{A}(j, \mathsf{X}) \Rightarrow j < i] \wedge \mathbb{C}(G_2)[j \in \mathbb{N}/\mathsf{A}(j, \mathsf{X}) \Rightarrow j \geq i]\right)$$

The constraint **(1)** for the call type has three events modeling (1) a call, (2) the start of the process and (3) the existence of the termination of the process. Moreover, the projected formulas hold at the configurations for these events. Every other event is a fEv. The exact position of termination (i.e., fEv events) is not specified in global types, so we do not constrain them. Reading from a location is defined analogously. The translation of $G_1.G_2$ models that there is a position i such that, for every object X, the events described in $\mathbb{C}(G_1)$ are in the subtrace before i and those in $\mathbb{C}(G_2)$ are in the subtrace after i.

The restriction is applied for every object, to ensure the following property: If a trace is a model for the translation of a type **G**, then for each participating object (1) all events of this objects have the same order as specified in **G** and (2) at the moment of the event, the corresponding FO formula holds. The translation of, e.g., $\mathsf{X}_1 \rightarrow \mathsf{X}_2 : \mathtt{m}_2.\mathsf{X}_1 \rightarrow \mathsf{X}_3 : \mathtt{m}_3$ describes that $\mathsf{X}_2.\mathtt{m}_2$ is called before $\mathsf{X}_2.\mathtt{m}_2$, but does *not* describe that the execution start in the same order. Thus, there are multiple possible event order satisfying this constraint, but from *every local point of view* the differences between these traces are not visible.

5 Analysis

Verifying deadlock freedom requires a *Points-To* analysis in addition to a type system. Deadlock freedom is equivalent to cycle-freedom of causality graphs [17] in Active Objects. The *causality graph* of a global type $\mathbf{G}$ is $\mathbb{G}(\mathbf{G}) = (V, E)$. Each node $L \in V$ is a local type, and each edge $(L_1, L_2) \in E$ models that L_2 must happen after L_1.

Definition 14 (Causality Graph). *Let* $\mathbf{G}$ *be a well-formed global type. The nodes of its causality graph* $\mathbb{G}(\mathbf{G})$ *are all partial local types derived from projecting* $\mathbf{G}$ *on all endpoints. An edge* (L_1, L_2) *is added if either (1)* $L_1 = L.L_2$ *is a partial type for some* L *in some projection on some object or (2)* L_1 *is the sending type and* L_2 *the receiving type from the projection of a single calling type.*

Note that global types do not contain sufficient information to deduce all causality, e.g., the causality of **get** statements cannot be deduced from a global type because synchronizations on futures are specified over *locations*.

We use a Points-To analysis for futures [17] instead. For generating a causality graph, we first derive a *partial* causality graph from the global type, and then we apply the Points-To analysis during type checking for the graph completion by deducing the missing edges. The Points-To analysis, defined below, determines which methods are responsible to resolve the futures in a given expression.

Definition 15 (Points-To). *The* Points-To analysis *determines the set* $\mathsf{p2}(\mathsf{e})$ *of methods, which may have resolved the future stored in an input expression* e*. We can express this using constraints, to integrate it into the type system:*

$$\forall i \in \mathbb{N}.\ \mathsf{C}(i) \doteq \mathbf{prc}(\mathsf{X}', f, \mathsf{val}(\mathsf{e}'), \sigma)\ \mathbf{prc}(\mathsf{X}, f', \mathtt{m}'(\mathtt{x} = \mathsf{e}.\mathbf{get}; \mathsf{s}''), \sigma'')\ \mathsf{C} \wedge [\![\mathsf{e}]\!]_{\sigma,\rho} = f \rightarrow$$
$$\exists j \in \mathbb{N}.\ j < i \wedge \mathsf{C}(j) \doteq \mathbf{prc}(\mathsf{X}', f, \mathtt{m}(\mathtt{s}), \sigma')\ \mathsf{C}' \wedge \mathtt{m} \in \mathsf{p2}(\mathsf{e})$$

Whenever a e.**get**-statement is checked against a type $\mathsf{Read}\ \mathsf{e}$, edges are added between the node of termination type of the methods which e can point to, and the node of the current type $\mathsf{Read}\ \mathsf{e}$. Although Points-To is undecidable, well-scaling tools which safely overapproximate are available [2].

Definition 16 (Admissibility). *A causality graph is* admissible *if (1) every path is cycle-free and (2) for every object* X*, and for any pair of receiving types of* X*, there exists a connecting path without an edge of the form* $(\mathsf{Put}\ \varphi, ?\mathtt{m}\langle\psi\rangle)$*.*

The graph on page 6 is admissible. With a non-admissible graph, methods may deadlock (violating (1)) or be executed in the wrong order (violating (2)).

Type System and Analysis. The auxiliary ADL-formula $\mathsf{post}(\mathsf{X}.\mathtt{m}, \varphi)$ models that the value in every future resolved by $\mathsf{X}.\mathtt{m}$ satisfies φ, while formula $\mathsf{Post}(\mathbf{G})$ represents the conjunction of all postconditions specified in $\mathbf{G}$. Figure 5 shows selected typing rules invoking the validity calculus [15] and Points-To analysis.

Before introducing the typing rules, we define $\mathsf{Roles}(\mathbf{G})$ as the set of objects in $\mathbf{G}$, $\mathbb{G}(\mathbf{G}) + E$ as the set of edges of $\mathbb{G}(\mathbf{G})$ and E (i.e., E is added into $\mathbb{G}(\mathbf{G})$), $\mathsf{term}(\mathtt{m})$ as the set of $\downarrow$ nodes of method $\mathtt{m}$, and $\mathsf{node}(\mathtt{s})$ as the set of nodes referring to the types that have typed $\mathtt{s}$. We define three kinds of type judgments:

(I) The Type Judgment for Programs. $\vdash \mathsf{Prgm} : \mathbf{G}$ checks Prgm against global type $\mathbf{G}$. The well-formedness of $\mathbf{G}$ (Definition 11) is ensured during type checking. Rule **(T-Main)** checks that every endpoint in $\mathbf{G}$ is implemented in Prgm, the main block makes the correct initializing call and checks each object against its object type. The edges collected from the typing rules for objects are added to the partial causality graph $\mathbb{G}(\mathbf{G})$ and the resulting graph is checked for admissibility.

(II) The Type Judgment for Objects. $\Phi \vdash \mathsf{O} : \mathbf{L} \rhd E$ checks whether O is well-typed by $\mathbf{L}$ under a given E with Φ. E is a set of causality edges and Φ is a set of ADL formulas. Rule **(T-Object)** projects $\mathbf{L}$ on all methods, checks each method $\mathtt{m}_i$ by $\mathbf{L} \upharpoonright \mathtt{m}_i$ and collects all resulting edges.

$$\textsf{(T-Main)}\ \frac{\begin{array}{c}\mathsf{O}_i = \mathtt{object}\ \mathsf{X}_i\{\dots\} \quad \mathsf{Roles}(\mathbf{G}) = \{\mathsf{X}_1,\dots,\mathsf{X}_n\} \qquad \mathbb{G}(\mathbf{G}) + \bigcup_{i\le n} E_i \text{ admissible}\\ \exists j \le n.\ \mathbf{G} = \mathtt{main}\!\rightarrow\!\mathsf{X}_\mathsf{j}\!:\!\mathtt{m}\langle\varphi\rangle.G \quad \forall i \le n.\ \mathsf{Post}(\mathbf{G}) \vdash \mathsf{O}_i : \mathsf{prp}^*(\mathbf{G} \upharpoonright \mathsf{X}_i) \rhd E_i\end{array}}{\vdash \mathsf{O}_1 \quad \dots \quad \mathsf{O}_n \quad \mathtt{main}\{\mathsf{X}_j!\mathtt{m}()\} : \mathbf{G}}$$

$$\textsf{(T-Object)}\ \frac{\begin{array}{c}\forall i \le n.\ \mathbf{L} \upharpoonright_{\mathsf{ac}} \mathtt{m}_i = ?\mathtt{m}_i\langle\varphi_i\rangle.L_i\\ \forall i \le n.\ \Phi, \varphi_i, \mathbf{skip} \vdash \mathtt{s}_i : L_i \rhd E_i \qquad E = \bigcup_{i\le n} E_i\end{array}}{\Phi \vdash \mathtt{object}\ \mathsf{X}\{\mathtt{T}_1\ \mathtt{m}_1(\overline{\mathtt{T}\ \mathtt{x}})\{\mathtt{s}_1\}\ \dots\ \mathtt{T}_n\ \mathtt{m}_n(\overline{\mathtt{T}\ \mathtt{x}})\{\mathtt{s}_n\}\quad \overline{\mathtt{T}\ \mathtt{x} = \mathtt{e}}\} : \mathbf{L} \rhd E}$$

$$\textsf{(T-Return)}\ \frac{\Phi \Rightarrow [\mathsf{s}; \mathbf{return}\ \mathsf{e}]\varphi}{\Phi, \mathsf{s} \vdash \mathbf{return}\ \mathsf{e} : \mathsf{Put}\ \varphi \rhd E} \qquad \textsf{(T-Call)}\ \frac{\begin{array}{c}\Phi, \mathsf{s}; \mathtt{T}\ \mathtt{x} = \mathsf{X}!\mathtt{m}(\bar{\mathsf{e}}) \vdash \mathsf{s}' : L \rhd E\\ \Phi \Rightarrow [\mathsf{s}; \mathtt{T}\ \mathtt{x} = \mathsf{X}!\mathtt{m}(\bar{\mathsf{e}})]\varphi\end{array}}{\Phi, \mathsf{s} \vdash \mathtt{T}\ \mathtt{x} = \mathsf{X}!\mathtt{m}(\bar{\mathsf{e}}); \mathsf{s}' : \mathsf{X}!_{\mathtt{X}}\mathtt{m}\langle\varphi\rangle.L \rhd E}$$

$$\textsf{(T-Get)}\ \frac{\begin{array}{c}\Phi, \mathsf{s}; \mathtt{T}\ \mathtt{x} = \mathsf{e}.\mathbf{get} \vdash \mathsf{s}' : L \rhd E'\\ E = E' \cup \{(n, n') \mid \exists \mathtt{m} \in \mathsf{p2}(\mathsf{e}).\ n \in \mathsf{term}(\mathtt{m}) \wedge n' \in \mathsf{node}(\mathsf{s}; \mathsf{e}.\mathbf{get})\}\end{array}}{\Phi, \mathsf{s} \vdash \mathtt{T}\ \mathtt{x} = \mathsf{e}.\mathbf{get}; \mathsf{s}' : \mathsf{Read}\ \mathsf{e}.L \rhd E}$$

Fig. 5. The selected typing rules.

(III) The Type Judgment for Statements. $\Phi, \mathtt{s} \vdash \mathtt{s} : L \rhd E$ checks whether $\mathtt{s}$ is well-typed by L under a given E with $\Phi, \mathtt{s}$. The environment $\mathtt{s}$ are the statements type-checked so far. Whenever an ADL formula is checked, a validity check is performed and $\mathtt{s}$ is added in the modality to consider the side-effects on the heap memory so far. However, these are not recorded in E: The causality edges only record which method a **get** statement synchronizes on. Rule **(T-Return)** checks that after executing all the type-checked statements, the **return** statement results in a state where φ holds. Rule **(T-Call)** also checks the formula φ which describes the state when the call has to be executed. Rule **(T-Get)** additionally executes the Points-To analysis and adds all the edges as described in the previous section.

Theorem 1 (Deadlock Freedom and Protocol Adherence). *Let* Prgm *be a program and* $\mathbf{G}$ *be a global type. If* Prgm *is well-typed against* $\mathbf{G}$ *then (1)* Prgm *does not deadlock and (2) every generated trace from* Prgm *satisfies* $\mathbb{C}(\mathbf{G})$*:*

$$\vdash \mathsf{Prgm} : \mathbf{G} \rightarrow \big(\forall \mathbf{tr}.\ \mathsf{Prgm} \Downarrow \mathbf{tr} \rightarrow \mathbf{tr} \models \mathbb{C}(\mathbf{G})\big)$$

6 Loops and Repetition

In this section we present the whole workflow of the previous section for Async extended with repetition. The language is extended with loops and the types with *repetition* types $(G)^*_\varphi$ (resp. $(L)^*_\varphi$). A repetition type resembles a Kleene-star and models the finite repetition of the type G (resp. L). The formula φ is a loop invariant and has to be satisfied whenever a loop iteration starts or ends.

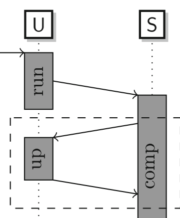
Definition 17 (Syntax with Repetition).

$$\mathtt{s} ::= \ldots \mid \textbf{while}(\mathtt{e})\{\mathtt{s}\} \qquad G ::= \ldots \mid (G)^*_\varphi.G \qquad L ::= \ldots \mid (L)^*_\varphi \,.\, L$$

By syntactic restrictions, the local type $\mathbf{L}$ of an object cannot have the form $(L)^*_\varphi.L'$, which forbids it to start with a loop. The intuition behind this restriction is that every loop has an invariant that an object must guarantee before executing the next iteration. If an object is not active before the loop, it cannot guarantee the invariant in the very beginning, thus repetition can start with the second action at the earliest. Below give an example for using invariants.

Example 4. Consider a big data analysis webtool with a client-side GUI U and a server-side computational server S. We model the following scenario:

U sends data to the computational server by calling `S.comp`. To stay responsive, U ends its initial process. U is called repeatedly on `G.up` by the server to update the progress. Whenever U is updated, the server also gets information by reading from the future of the last call to `U.up`. The sequence diagram to the right illustrates the protocol. During updating, U must stay in a state expecting to receive updates from the server. It is therefore important to specify that field `U.expect` is not `Nil`.



$$\mathbf{main} \to \mathsf{U}{:}\mathsf{run}\langle\top\rangle \,.\, \mathsf{U} \to \mathsf{S}{:}\mathtt{comp}\langle\top,\top\rangle \,.\, \left(\mathsf{S} \xrightarrow{\mathtt{x}} \mathsf{U}{:}\mathtt{up}\langle\top,\top\rangle \,.\, \mathsf{S}\uparrow\mathtt{x}\right)^*_{\mathtt{U.expect}\neq\mathtt{Nil}} \,.\, \mathbf{end}$$

The invariant $\mathsf{U}.\mathtt{expect} \neq \mathtt{Nil}$ specifies the condition that field `U.expect` is a non-empty list. This is propagated during projection, which results in the following local type for `U.up`

$$?\mathtt{update}\langle\mathbf{self}.\mathtt{expect} \neq \mathtt{Nil}\rangle.\mathsf{Put}\ \mathbf{self}.\mathtt{expect} \neq \mathtt{Nil}$$

There is no repetition because *being repeatedly called* is only visible for the whole object, not a single process. The type of `S.cmp` however contains a repetition:

$$?\mathtt{comp}\langle\top\rangle.\left(\mathtt{U}!_{\mathtt{x}}\mathtt{up}\langle\top\rangle.\mathsf{Read}\ \mathtt{x}\right)_{\exists l.\ l\neq\mathtt{Nil}}.\mathsf{Put}\ \top$$

The workflow is the same as described above. We provide the projection, translation, propagation and typing rules as extension of the previous systems.

Definition 18 (Projection Rules for Loops). *The auxiliary predicate* $rcv(\mathsf{X}, G)$ *holds if* X *is specified as being called in* G.

$$\left((G)^*_\varphi.G'\right)\restriction_{\mathsf{ac}} \mathsf{X} = \begin{cases} \mathsf{Put}\ \mathsf{ac}(\mathsf{X}).(L'')^*_{\varphi @ \mathsf{X}}.L' & \textit{if } L \neq \mathsf{skip} \wedge \quad rcv(\mathsf{X}, G) \wedge \mathsf{ac}(\mathsf{X}) \neq \bot \wedge cs(\varphi) \\ (L)^*_{\varphi @ \mathsf{X}}.L' & \textit{if } L \neq \mathsf{skip} \wedge \neg rcv(\mathsf{X}, G) \wedge \mathsf{ac}(\mathsf{X}) \neq \bot \wedge cs(\varphi) \\ L' & \textit{if } L = \mathsf{skip} \wedge cs(\varphi) \end{cases}$$

$$\textit{Where } G.\mathsf{End}\restriction_{\mathsf{ac}[\mathsf{X}\mapsto\bot]} \mathsf{X} = L'',\ G\restriction_{\mathsf{ac}} \mathsf{X} = L \textit{ and } G'\restriction_{\mathsf{ac}} \mathsf{X} = L'$$

The auxiliary formula $cs(\varphi)$ specifies that all weakenings of φ imply φ. This is necessary to reject invariants that connect multiple heaps: e.g., this condition would reject $\texttt{G.i} \doteq \texttt{S.i}$, as it cannot be guaranteed by `G` and `S` separately. This condition, however, admits $\texttt{G.i} \doteq 1 \wedge \texttt{S.i} = 1$. The first rule projects global types to object types. The first case is applied if the object participates in the repetition of the inner type G *by being repeatedly called*. The last active process must terminate first and the repeatedly called method must terminate within the repetition. The termination inside the loop is ensured by projecting the inner type with an appended End. The second case is applied if the object participates in the repetition ($L \neq \mathsf{skip}$) by any other repeated action then being called ($\neg rcv(\mathsf{X}, G)$). Finally, the last case skips the repetition if the object does not participate in it.

The second rule projects object types to methods. The rule distinguishes whether the whole process is inside the repetition or not. If the process is completely inside, the repetition is removed, as it is not visible to the method.

In presence of repetition, invariants have to be propagated inside the repeated, the previous, and the next types. The following definition summarizes gives the rules for repetition, additionally to rule **(1)** in Definition 10.

Definition 19 (Rules for Propagation for Repetition).

(2) $\mathsf{Put}\ \varphi.(L)^*_\psi \rightsquigarrow \mathsf{Put}\ \varphi \wedge \psi.(L)^*_\psi$ *(3)* $(L)^*_\psi.?\mathtt{m}'\langle\varphi\rangle \rightsquigarrow (L)^*_\psi.?\mathtt{m}'\langle\varphi \wedge \psi\rangle$

(4) $(L)^*_\varphi.(L)^*_\psi \rightsquigarrow (L)^*_{\varphi\wedge\psi}.(L)^*_\psi$ *(5)* $(?\mathtt{m}'\langle\varphi\rangle.L.\mathsf{Put}\ \varphi')^*_\psi \rightsquigarrow (?\mathtt{m}'\langle\varphi \wedge \psi\rangle.L.\mathsf{Put}\ \varphi' \wedge \psi)^*_\psi$

Since loop invariants have to hold *before* the first repetition, rule **(2)** ensures that the last process before a repetition satisfies the invariant when terminating. Rule **(3)** adds an invariant to the next process, as the invariant also holds *after* the last repetition. Rule **(4)** is another case of the first one, in case two repetitions are succeeding each other. Finally, rule **(5)** adds the invariant to the processes inside the repetition. This rule enables the use of the invariant in the first method of the repetition and ensures that the last method reestablishes the invariant.

For the translation into constraints, first-order constraints are not expressive enough. The Kleene star constraint resembles regular languages and we thus use *monadic second order logic* (MSO) to capture repetition. MSO extends first-order logic with a quantifier $\exists Y \subseteq Z$ which quantities over subsets of Z and a $\in$ primitive to express membership of those sets. The extension of relativization is straightforward [23,30]. We now extend the semantics of types as constraints from Definition 13 to repetition:

Definition 20 (Semantics of Repetition). *The semantics of repeated types uses a set of boundary indices X, between which the inner translation must. Also, the invariant has to hold at every boundary.*

$$\mathbb{C}\big((G^*_\varphi)\big) \;=\; \exists X \subseteq \mathbb{N}.\ \exists i,j \in X.\ \big(\forall k \in \mathbb{N}.\ i < k \leq j\big) \wedge \forall i \in X.\ \mathsf{C}(i) \models \varphi \wedge$$
$$\forall i,j \in X.\ \Big(\big(\forall k \in X.\ k \geq j \vee k \leq i\big) \Rightarrow \big(\mathbb{C}(G)\big)[n \in \mathbb{N}/i < n \leq j]\Big)$$

The typing rule for repetition resembles invariant rules from Hoare calculi [26]:

$$\textsf{(T-While)}\quad \frac{\varphi \wedge \mathsf{Post}(\mathbf{G}), \mathbf{skip} \vdash \mathtt{s}' : L' \rhd E'' \qquad \Phi \Rightarrow [\mathtt{s}'']\varphi \qquad \varphi \wedge \mathsf{Post}(\mathbf{G}) \Rightarrow [\mathtt{s}]\varphi \qquad \varphi \wedge \mathsf{Post}(\mathbf{G}), \mathbf{skip} \vdash \mathtt{s} : L \rhd E' \qquad E = E' \cup E''}{\Phi, \mathtt{s}'' \vdash \mathbf{while}\ e\ \{\mathtt{s}\}; \mathtt{s}' : (L)^*_\varphi\ .\ L' \rhd E}$$

The first premise continues the type checking of the program, in an environment where only the information in the invariant (and the global information in Post, as defined in Sect. 5) is available. The second and third premises check that the invariant holds initially and is preserved by the loop body. The forth premise checks the loop body and the last premise combines the derived causality edges. The extension of the causality graph is described in [29].

Corollary 1. *Theorem 1 holds for the system with repetition.*

7 Conclusion and Related Work

In this paper we generalize MPST for Active Objects to a two-phase analysis that handles protocols where information is not only transmitted between objects via asynchronous method calls but also inside the object through the heap memory of Active Objects. Additionally, we provide a model-theoretic semantics for MPST, which allows us to give a declarative definition of protocol adherence and integrate further static analyses. These analyses are used to reason about method order and future synchronization within a type system.

7.1 Discussion

Decidability and Types for Validation. The judgment $\vdash \mathsf{Prgm} : \mathbf{G}$ is undecidable if the validity of the FO logic used for specifying side-effects is undecidable. A developer can choose an FOL fragment with decidable validity to trade off expressiveness against analyzability, e.g., if the developer chooses a more restricted fragment, which may limit the expressiveness of the specification, then the validity of the FO logic used for specifying side-effects may become decidable.

When using an undecidable FOL fragment, our approach can be used as a *validation* tool to check whether the implemented (sub-)system will be behaving as expected. Our approach can be integrated into the development process similarly as invariant-based approaches, and applies techniques proposed by MPST to connect global and local views of concurrent programs, a notoriously difficult problem when using contracts and invariants [15].

Protocol Adherence. Current work on MPST defines protocol adherence as a fidelity theorem, which states that every sequence of interactions in a session follows the scenario declared in MPST [27] as follows: An operational semantics for types is defined and it is shown that the semantics of the language is a refinement of the semantics of the types. Similarly, behavioral contracts [10] define protocol adherence by *compliance*, which compares the interaction of contracts. These are *operational* approaches to specification. We define protocol adherence from a *declarative* perspective by requiring a logical *property* to hold for all traces of a well-typed program. A declarative specification can be analyzed with tools for logical specification, and can enable easier integration of other static analysis tools (e.g., to consider state), because they are only required to have a logical characterization.

7.2 Related Work

This work extends our previous system for Active Objects [31], which could not specify and verify state, required an additional verification step for the scheduler and explicit termination points within the global type.

Actors and Objects. Crafa and Padovani [11,35] investigate behavioral types for the object-oriented join calculus with typestate, a concurrency model similar to actors. Gay et al. [18] model channels as objects, integrating MPST with classes; Dezani-Ciancaglini et al. [13] use MPST in the object-oriented language `MOOSE`, where types describe communication through shared channels. We ensure deadlock freedom similarly to Giachino et al. [20,21], who ensure deadlock freedom by inferring behavioral *contracts* and applying a cycle detection algorithm; however, they do not consider protocol adherence.

State and Contracts. Bocchi et al. [5–7] develop a MPST discipline with assertions for endpoint state. The work considers neither objects nor heap memory. The specifications use *global values* in global types and require complex checks for *history-sensitivity* and *temporal-sensitivity* to ensure that an endpoint proves its obligations. We evade this by specifying inherently class-local memory *locations*. They explicitly track values over several endpoints, while we implicitly do so by equations over locations. In a stateless setting, Toninho and Yoshida use dependent MPST [38] to reason about passed data.

Logics. Session types as formulas have been examined by Caires et al. [8] and Carbone et al. [9] for intuitionistic and linear logics as types-as-proposition for the π-calculus. Our work uses logic not for a *proof-theoretic* types-as-proposition theorem, but to use a *model-theoretic* notion of protocol adherence and to integrate static analysis and dynamic logic. Lange and Yoshida [33] also characterize session types as formulas, but their characterization characterizes the *subtyping* relation, not the execution traces as in our work.

Acknowledgments. This work is partially supported by `FormbaR`, part of the Innovation Alliance between TU Darmstadt and Deutsche Bahn AG.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice. LNCS, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Albert, E., Flores-Montoya, A., Genaim, S., Martin-Martin, E.: May-happen-in-parallel analysis for actor-based concurrency. ACM Trans. Comput. Log. **17**(2), 11 (2016)
3. Ancona, D., Bono, V., Bravetti, M.: Behavioral Types in Programming Languages. Now Publishers Inc., Hanover (2016)
4. Baker, H.G., Hewitt, C.: The incremental garbage collection of processes. SIGART Newsl. **64**, 55–59 (1977)
5. Bocchi, L., Demangeon, R., Yoshida, N.: A multiparty multi-session logic. In: Palamidessi, C., Ryan, M.D. (eds.) TGC 2012. LNCS, vol. 8191, pp. 97–111. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41157-1_7
6. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 162–176. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15375-4_12
7. Bocchi, L., Lange, J., Tuosto, E.: Three algorithms and a methodology for amending contracts for choreographies. Sci. Ann. Comp. Sci. **22**(1), 61–104 (2012)
8. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 222–236. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15375-4_16
9. Carbone, M., Lindley, S., Montesi, F., Schürmann, C., Wadler, P.: Coherence generalises duality: a logical explanation of multiparty session types. In: CONCUR 2016. LIPIcs, vol. 59, pp. 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
10. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. ACM Trans. Program. Lang. Syst. **31**(5), 19:1–19:61 (2009)
11. Crafa, S., Padovani, L.: The chemical approach to typestate-oriented programming. ACM Trans. Program. Lang. Syst. **39**(3), 13:1–13:45 (2017)
12. de Boer, F.S., et al.: A survey of active object languages. ACM Comput. Surv. **50**(5), 76:1–76:39 (2017)
13. Dezani-Ciancaglini, M., Drossopoulou, S., Mostrous, D., Yoshida, N.: Objects and session types. Inf. Comput. **207**(5), 595–641 (2009)
14. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: a deductive verification tool for the concurrent modelling language ABS. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 517–526. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_35
15. Din, C.C., Owe, O.: A sound and complete reasoning system for asynchronous communication with shared futures. J. Log. Algebr. Meth. Program. **83**(5–6), 360–383 (2014)
16. Din, C.C., Tapia Tarifa, S.L., Hähnle, R., Johnsen, E.B.: History-based specification and verification of scalable concurrent and distributed systems. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 217–233. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25423-4_14

17. Flores-Montoya, A.E., Albert, E., Genaim, S.: May-happen-in-parallel based deadlock analysis for concurrent objects. In: Beyer, D., Boreale, M. (eds.) FMOODS/-FORTE -2013. LNCS, vol. 7892, pp. 273–288. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38592-6_19
18. Gay, S.J., Gesbert, N., Ravara, A., Vasconcelos, V.T.: Modular session types for objects. Log. Methods Comput. Sci. **11**(4) (2015)
19. Gay, S.J., Vasconcelos, V.T., Wadler, P., Yoshida, N.: Theory and applications of behavioural types (dagstuhl seminar 17051). Dagstuhl Reports **7**(1), 158–189 (2017)
20. Giachino, E., Henrio, L., Laneve, C., Mastandrea, V.: Actors may synchronize, safely! In: PPDP, pp. 118–131. ACM (2016)
21. Giachino, E., Laneve, C., Lienhardt, M.: A framework for deadlock detection in core ABS. Softw. Syst. Model. **15**(4), 1013–1048 (2016)
22. Harel, D.: First-Order Dynamic Logic. Springer, Secaucus (1979). https://doi.org/10.1007/3-540-09237-4
23. Henkin, L.: Relativization with respect to formulas and its use in proofs of independence. Compositio Mathematica **20**, 88–106 (1968)
24. Henrio, L., Laneve, C., Mastandrea, V.: Analysis of synchronisations in stateful active objects. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 195–210. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_13
25. Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI 1973, pp. 235–245. Morgan Kaufmann Publishers Inc. (1973)
26. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
27. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 1–67 (2016)
28. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
29. Kamburjan, E.: Session types for ABS. Technical report, TU Darmstadt (2016). http://formbar.raillab.de/en/techreportsessiontypesabs/
30. Kamburjan, E., Chen, T.: Stateful behavioral types for ABS. CoRR, abs/1802.08492 (2018)
31. Kamburjan, E., Din, C.C., Chen, T.-C.: Session-based compositional analysis for actor-based languages using futures. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM 2016. LNCS, vol. 10009, pp. 296–312. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47846-3_19
32. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
33. Lange, J., Yoshida, N.: Characteristic formulae for session types. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 833–850. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_52
34. Odersky, M., et al.: Scala Programming Language. http://www.scala-lang.org
35. Padovani, L.: Deadlock-free typestate-oriented programming. Submitted to The Art, Science, and Engineering of Programming (2017, preprint). https://hal.archives-ouvertes.fr/hal-01628801/file/main.pdf

36. Schmitt, P.H., Ulbrich, M., Weiß, B.: Dynamic frames in java dynamic logic. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 138–152. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18070-5_10
37. Tasharofi, S., Dinges, P., Johnson, R.E.: Why do scala developers mix the actor model with other concurrency models? In: Castagna, G. (ed.) ECOOP 2013. LNCS, vol. 7920, pp. 302–326. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39038-8_13
38. Toninho, B., Yoshida, N.: Certifying data in multiparty session types. J. Log. Algebr. Meth. Program. **90**, 61–83 (2017)

Probabilistic Verification of Timing Constraints in Automotive Systems Using UPPAAL-SMC

Eun-Young Kang[1,2(✉)], Dongrui Mu[2], and Li Huang[2]

[1] University of Namur, Namur, Belgium
eykang@unamur.be
[2] School of Data and Computer Science, Sun Yat-Sen University, Guangzhou, China
{mudr,huangl223}@mail2.sysu.edu.cn

Abstract. Modeling and analysis of non-functional properties, such as timing constraints, is crucial in automotive real-time embedded systems. East-adl is a domain specific architectural language dedicated to safety-critical automotive embedded system design. We have previously specified East-adl timing constraints in Clock Constraint Specification Language (Ccsl) and proved the correctness of specification by mapping the semantics of the constraints into Uppaal models amenable to model checking. In most cases, a bounded number of violations of timing constraints in automotive systems would not lead to system failures when the results of the violations are negligible, called Weakly-Hard (WH). Previous work is extended in this paper by including support for probabilistic analysis of timing constraints in the context of WH: Probabilistic extension of Ccsl, called PrCcsl, is defined and the East-adl timing constraints with stochastic properties are specified in PrCcsl. The semantics of the extended constraints in PrCcsl is translated into Uppaal-SMC models for formal verification. Furthermore, a set of mapping rules is proposed to facilitate guarantee of translation. Our approach is demonstrated on an autonomous traffic sign recognition vehicle case study.

Keywords: East-adl · Uppaal-SMC · Probabilistic Ccsl
Weakly-Hard System · Statistical model checking

1 Introduction

Model-driven development is rigorously applied in automotive systems in which the software controllers interact with physical environments. The continuous time behaviors (evolved with various energy rates) of those systems often rely on complex dynamics as well as on stochastic behaviors. Formal verification and validation (V&V) technologies are indispensable and highly recommended for development of safe and reliable automotive systems [3,4]. Conventional V&V, i.e., testing and model checking have limitations in terms of assessing the reliability of hybrid systems due to both the stochastic and non-linear dynamical

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 236–254, 2018.
https://doi.org/10.1007/978-3-319-98938-9_14

features. To ensure the reliability of safety critical hybrid dynamic systems, *statistical model checking (SMC)* techniques have been proposed [11,12,25]. These techniques for fully stochastic models validate probabilistic performance properties of given deterministic (or stochastic) controllers in given stochastic environments.

Conventional formal analysis of timing models addresses worst case designs, typically used for hard deadlines in safety critical systems, however, there is great incentive to include "less-than-worst-case" designs to improve efficiency but without affecting the quality of timing analysis in the systems. The challenge is the definition of suitable model semantics that provide reliable predictions of *system timing*, given the timing of individual components and their compositions. While the standard worst case models are well understood in this respect, the behavior and the expressiveness of "less-than-worst-case" models is far less investigated. In most cases, a bounded number of violations of timing constraints in systems would not lead to system failures when the results of the violations are negligible, called Weakly-Hard (WH) [8,29]. In this paper, we propose a formal probabilistic modeling and analysis technique by extending the known concept of WH constraints to what is called "typical" worst case model and analysis.

East-adl (Electronics Architecture and Software Technology - Architecture Description Language) [5,14], aligned with AUTOSAR (Automotive Open System Architecture) standard [1], is a concrete example of the MBD approach for the architectural modeling of safety-critical automotive embedded systems. A system in East-adl is described by `Functional Architectures (FA)` at different abstraction levels. The `FA` are composed of a number of interconnected *functionprototypes* (f_p), and the f_ps have ports and connectors for communication. East-adl relies on external tools for the analysis of specifications related to requirements. For example, behavioral description in East-adl is captured in external tools, i.e., Simulink/Stateflow[32]. The latest release of East-adl has adopted the time model proposed in the Timing Augmented Description Language (Tadl2) [9]. Tadl2 expresses and composes the basic timing constraints, i.e., repetition rates, end-to-end delays, and synchronization constraints. The time model of Tadl2 specializes the time model of MARTE, the UML profile for Modeling and Analysis of Real-Time and Embedded systems [30]. MARTE provides Ccsl, a time model and a Clock Constraint Specification Language, that supports specification of both logical and dense timing constraints for MARTE models, as well as functional causality constraints [27].

We have previously specified non-functional properties (timing and energy constraints) of automotive systems specified in East-adl and MARTE/Ccsl, and proved the correctness of specification by mapping the semantics of the constraints into Uppaal models for model checking [23]. Previous work is extended in this paper by including support for probabilistic analysis of timing constraints of automotive systems in the context WH: 1. Probabilistic extension of Ccsl, called PrCcsl, is defined and the East-adl/Tadl2 timing constraints with stochastic properties are specified in PrCcsl; 2. The semantics of the extended constraints in PrCcsl is translated into verifiable Uppaal-SMC [2] models for

formal verification; 3. A set of mapping rules is proposed to facilitate guarantee of translation. Our approach is demonstrated on an autonomous traffic sign recognition vehicle (AV) case study.

The paper is organized as follows: Sect. 2 presents an overview of CCSL and UPPAAL-SMC. The AV is introduced as a running example in Sect. 3. Section 4 presents the formal definition of PrCCSL. Section 5 describes a set of translation patterns from CCSL/PrCCSL to UPPAAL-SMC models and how our approaches provide support for formal analysis at the design level. The applicability of our method is demonstrated by performing verification on the AV case study in Sect. 6. Sections 7 and 8 present related work and the conclusion.

2 Preliminary

In our framework, we consider a subset of CCSL and its extension with stochastic properties that is sufficient to specify EAST-ADL timing constraints in the context of WH. Formal Modeling and V&V of the EAST-ADL timing constraints specified in CCSL are performed using UPPAAL-SMC.

Clock Constraint Specification Language (CCSL) [6,27] is a UML profile for modeling and analysis of real-time systems (MARTE) [7,26]. In CCSL, a clock represents a sequence of (possibly infinite) instants. An event is a clock and the occurrences of an event correspond to a set of ticks of the clock. CCSL provides a set of clock constraints that specifies evolution of clocks' ticks. The physical time is represented by a dense clock with a base unit. A dense clock can be discretized into a discrete/logical clock. *idealClock* is a predefined dense clock whose unit is second. We define a universal clock *ms* based on *idealClock*: *ms* = *idealClock* `discretizedBy` 0.001. *ms* representing a periodic clock that ticks every 1 ms in this paper. A step is a tick of the universal clock. Hence the length of one step is 1 ms.

CCSL provides two types of clock constraints, *relation* and *expression*: A *relation* limits the occurrences among different events/clocks. Let C be a set of clocks, $c1, c2 \in C$, `coincidence` relation ($c1 \equiv c2$) specifies that two clocks must tick simultaneously. `Precedence` relation ($c1 \prec c2$) delimits that $c1$ runs faster than $c2$, i.e., $\forall k \in \mathbb{N}^+$, where $\mathbb{N}^+$ is the set of positive natural numbers, the kth tick of $c1$ must occur prior to the kth tick of $c2$. `Causality` relation ($c1 \preceq c2$) represents a relaxed version of `precedence`, allowing the two clocks to tick at the same time. `Subclock` ($c1 \subseteq c2$) indicates the relation between two clocks, *superclock* ($c1$) and *subclock* ($c2$), s.t. each tick of the subclock must correspond to a tick of its superclock at the same step. `Exclusion` ($c1 \# c2$) prevents the instants of two clocks from being coincident. An *expression* derives new clocks from the already defined clocks: `periodicOn` builds a new clock based on a *base* clock and a *period* parameter, s.t., the instants of the new clocks are separated by a number of instants of the *base* clock. The number is given as *period*. `DelayFor` results in a clock by delaying the *base* clock for a given number of ticks of a *reference* clock. `Infimum`, denoted `inf`, is defined as the slowest clock that is

faster than both $c1$ and $c2$. `Supremum`, denoted `sup`, is defined as the fastest clock that is slower than $c1$ and $c2$.

UPPAAL-SMC performs the probabilistic analysis of properties by monitoring simulations of complex hybrid systems in a given stochastic environment and using results from the statistics to determine whether the system satisfies the property with some degree of confidence. Its clocks evolve with various rates, which are specified with *ordinary differential equations* (ODE). UPPAAL-SMC provides a number of queries related to the stochastic interpretation of Timed Automata (STA) [12] and they are as follows, where N and *bound* indicate the number of simulations to be performed and the time bound on the simulations respectively:

1. *Probability Estimation* estimates the probability of a requirement property ϕ being satisfied for a given STA model within the time bound: $Pr[bound]\ \phi$.
2. *Hypothesis Testing* checks if the probability of ϕ being satisfied is larger than or equal to a certain probability P_0: $Pr[bound]\ \phi\ \geqslant\ P_0$.
3. *Probability Comparison* compares the probabilities of two properties being satisfied in certain time bounds: $Pr[bound_1]\ \phi_1 \geqslant Pr[bound_2]\ \phi_2$.
4. *Expected Value* evaluates the minimal or maximal value of a clock or an integer value while UPPAAL-SMC checks the STA model: $E[bound; N](min : \phi)$ or $E[bound; N](max : \phi)$.
5. *Simulations*: UPPAAL-SMC runs N simulations on the STA model and monitors k (state-based) properties/expressions $\phi_1, ..., \phi_k$ along the simulations within simulation bound *bound*: $simulate\ N\ [\leqslant bound]\{\phi_1, ..., \phi_k\}$.

3 Running Example: Traffic Sign Recognition Vehicle

An autonomous vehicle (AV) [21,22] application using Traffic Sign Recognition is adopted to illustrate our approach. The AV reads the road signs, e.g., "speed limit" or "right/left turn", and adjusts speed and movement accordingly. The functionality of AV, augmented with timing constraints and viewed as `Functional Design Architecture (FDA)` (`designFunctionTypes`), consists of the following f_ps in Fig. 1: `System` function type contains four f_ps, i.e., the `Camera` captures sign images and relays the images to `SignRecognition` periodically. `Sign Recognition` analyzes each frame of the detected images and computes the desired images (sign types). `Controller` determines how the speed of the vehicle is adjusted based on the sign types and the current speed of the vehicle. `VehicleDynamic` specifies the kinematics behaviors of the vehicle. `Environment` function type consists of three f_ps, i.e., the information of traffic signs, random obstacles, and speed changes caused by environmental influence described in `TrafficSign`, `Obstacle`, and `Speed` f_ps respectively.

We consider the `Periodic, Execution, End-to-End, Synchronization, Sporadic, and Comparison` timing constraints on top of the AV EAST-ADL model, which are sufficient to capture the constraints described in Fig. 1. Furthermore, we extend EAST-ADL/TADL2 with an `Exclusion` timing constraint

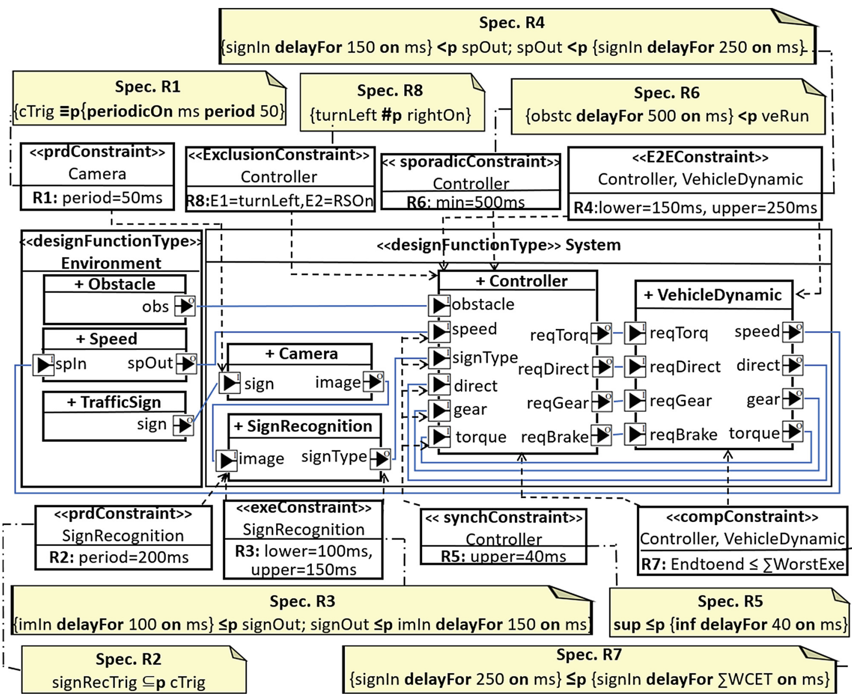


Fig. 1. AV in EAST-ADL augmented with TADL2 constraints (`R. IDs`) specified in PrCCSL (`Spec. R. IDs`)

(R8 in Fig. 1) that integrates relevant concepts from the CCSL constraint, i.e., two events cannot occur simultaneously.

R1. The camera must capture an image every 50 ms. In other words, a `Periodic` acquisition of `Camera` must be carried out every 50 ms.
R2. The captured image must be recognized by an AV every 200 ms, i.e., a `Periodic` constraint on `SignRecognition` f_p.
R3. The detected image should be computed within [100, 150] ms in order to generate the desired sign type, the `SignRecognition` must complete its execution within [100, 150] ms.
R4. When a traffic sign is recognized, the speed of AV should be updated within [150, 250] ms. An `End-to-End` constraint on `Controller` and `VehicleDynamic`, i.e., the time interval from the input of `Controller` to the output of `VehicleDynamic` must be within a certain time.
R5. The required environmental information should arrive to the controller within 40 ms. Input signals (`speed`, `signType`, `direct`, `gear` and `torque` ports) must be detected by `Controller` within a given time window, i.e., the tolerated maximum constraint is 40 ms.
R6. If the mode of AV switches to "emergency stop" due to a certain obstacle, it should not revert back to "automatic running" mode within a specific time

period. It is interpreted as a `Sporadic` constraint, i.e., the mode of AV is changed to `Stop` because of encountering an obstacle, it should not revert back to `Run` mode within 500 ms.
R7. The execution time interval from `Controller` to `VehicleDynamic` must be less than or equal to the sum of the worst case execution time interval of each f_p.
R8. While AV turns left, the “turning right” mode should not be activated. The events of turning left and right considered as exclusive and specified as an `Exclusion` constraint.

`Delay` constraint gives duration bounds (minimum and maximum) between two events *source* and *target*. This is specified using *lower, upper* values given as either `Execution` constraint (R3) or `End-to-End` constraint (R4). `Synchronization` constraint describes how tightly the occurrences of a group of events follow each other. All events must occur within a sliding window, specified by the *tolerance* attribute, i.e., the maximum time interval allowed between events (R5). `Periodic` constraint states that the period of successive occurrences of a single event must have a time interval (R1–R2). `Sporadic` constraint states that *events* can arrive at arbitrary points in time, but with defined minimum inter-arrival times between two consecutive occurrences (R6). `Comparison` constraint delimits that two consecutive occurrences of an event should have a minimum inter-arrival time (R7). `Exclusion` constraint refers that two events must not occur at the same time (R8).

Those timing constraints are formally specified (see as `R. IDs` in Fig. 1) using the subset of clock *relations* and *expressions* (see Sect. 2) in the context of WH. The timing constraints are then verified utilizing UPPAAL-SMC and are described further in the following sections.

4 Probabilistic Extension of *Relation* in CCSL

To perform the formal specification and probabilistic verification of EAST-ADL timing constraints (R1–R8 in Sect. 3.), CCSL *relations* are augmented with probabilistic properties, called PrCCSL, based on WH [8]. More specifically, in order to describe the bound on the number of permitted timing constraint violations in WH, we extend CCSL *relations* with a probabilistic parameter p, where p is the probability threshold. PrCCSL is satisfied if and only if the probability of *relation* constraint being satisfied is greater than or equal to p. As illustrated in Fig. 1, EAST-ADL/TADL2 timing constraints (R. IDs in Fig. 1) can be specified (Spec. R. IDs) using the PrCCSL *relations* and the conventional CCSL *expressions*.

A *time system* is specified by a set of clocks and clock constraints. An execution of the time system is a `run` where the occurrences of events are clock ticks.

Definition 1 (`Run`). *A* `run` *R consists of a finite set of consecutive steps where a set of clocks tick at each step i. The set of clocks ticking at step i is denoted as $R(i)$, i.e., for all i, $0 \leqslant i \leqslant n$, $R(i) \in R$, where n is the number of steps of R.*

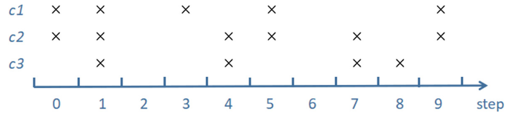


Fig. 2. Example of a Run

Figure 2 presents a `run` R consisting of 10 steps and three clocks $c1$, $c2$ and $c3$. The ticks of the three clocks along with steps are shown as "cross" symbols (x). For instance, $c1$, $c2$ and $c3$ tick at the first step, hence $R(1) = \{c1, c2, c3\}$.

The history of a clock c presents the number of times the clock c has ticked prior to the current step.

Definition 2 (`History`). *For $c \in C$, the* `history` *of c in a run R is a function: H_R^c: $\mathbb{N} \to \mathbb{N}$. For all instances of step i, $i \in \mathbb{N}$, $H_R^c(i)$ indicates the number of times the clock c has ticked prior to step i in run R, which is initialized as 0 at step 0. It is defined as:*

$$H_R^c(i) = \begin{cases} 0, & i = 0 \\ H_R^c(i-1), & c \notin R(i) \wedge i > 0 \\ H_R^c(i-1) + 1, & c \in R(i) \wedge i > 0 \end{cases}$$

Definition 3 (`PrCCSL`). *Let $c1$, $c2$ and R be two logical clocks and a* `run`*. The probabilistic extension of relation constraints, denoted $c1{\sim_p}c2$, is satisfied if the following condition holds:*

$$R \vDash c1{\sim_p}c2 \iff Pr(c1{\sim}c2) \geqslant p$$

where $\sim \in \{\subseteq, \equiv, \prec, \preceq, \#\}$, $Pr(c1{\sim}c2)$ is the probability of the relation $c1{\sim}c2$ being satisfied, and p is the probability threshold.

The five CCSL *relations*, `subclock`, `coincidence`, `exclusion`, `causality` and `precedence`, are considered and their probabilistic extensions are defined.

Definition 4 (`Probabilistic Subclock`). *Let $c1$, $c2$ and $\mathcal{M}$ be two logical clocks and a system model. Given k* `runs` *$= \{R_1, \ldots, R_k\}$, the probabilistic extension of* `subclock` *relation between $c1$ and $c2$, denoted $c1{\subseteq_p}c2$, is satisfied if the following condition holds:*

$$\mathcal{M} \vDash c1{\subseteq_p}c2 \iff Pr[c1{\subseteq}c2] \geqslant p$$

where $Pr[c1{\subseteq}c2] = \frac{1}{k} \sum_{j=1}^{k} \{R_j \models c1{\subseteq}c2\}$, $R_j \in \{R_1, \ldots, R_k\}$, i.e., the ratio of `runs` *that satisfies the* `subclock` *relation out of k* `runs`*.*

A `run` R_j satisfies the `subclock` relation between $c1$ and $c2$ "if $c1$ ticks, $c2$ must tick" holds at every step i in R_j, s.t., $(R_j \models c1{\subseteq}c2) \iff (\forall i\ 0 \leqslant i \leqslant n,\ c1 \in$

$R(i) \implies c2 \in R(i))$. "$R_j \models c1 \subseteq c2$" returns 1 if R_j satisfies $c1 \subseteq c2$, otherwise it returns 0.

`Coincidence` relation delimits that two clocks must always tick at the same step, i.e., if $c1$ and $c2$ are coincident, then $c1$ and $c2$ are subclocks of each other.

Definition 5 (`Probabilistic Coincidence`). *The probabilistic* ***`coincidence`*** *relation between* $c1$ *and* $c2$*, denoted* $c1 \equiv_p c2$*, is satisfied over* $\mathcal{M}$ *if the following condition holds:*

$$\mathcal{M} \models c1 \equiv_p c2 \iff Pr[c1 \equiv c2] \geqslant p$$

where $Pr[c1 \equiv c2] = \frac{1}{k} \sum_{j=1}^{k} \{R_j \models c1 \equiv c2\}$ *is determined by the number of* ***`runs`*** *satisfying the* ***`coincidence`*** *relation out of* k ***`runs`****.*

A `run`, R_j satisfies the `coincidence` relation on $c1$ and $c2$ if the assertion holds: $\forall i,\ 0 \leqslant i \leqslant n,\ (c1 \in R(i) \implies c2 \in R(i)) \wedge (c2 \in R(i) \implies c1 \in R(i))$. In other words, the satisfaction of `coincidence` relation is established when the two conditions "if $c1$ ticks, $c2$ must tick" and "if $c2$ ticks, $c1$ must tick" hold at every step.

The inverse of `coincidence` relation is `exclusion`, which specifies two clocks cannot tick at the same step.

Definition 6 (`Probabilistic Exclusion`). *For all* k ***`runs`*** *over* $\mathcal{M}$*, the probabilistic* ***`exclusion`*** *relation between* $c1$ *and* $c2$*, denoted* $c1 \#_p c2$*, is satisfied if the following condition holds:*

$$\mathcal{M} \models c1 \#_p c2 \iff Pr[c1 \# 2] \geqslant p$$

where $Pr[c1 \# c2] = \frac{1}{k} \sum_{j=1}^{k} \{R_j \models c1 \# c2\}$ *is the ratio of the* ***`runs`*** *satisfying the* ***`exclusion`*** *relation out of* k ***`runs`****.*

A `run`, R_j, satisfies the `exclusion` relation on $c1$ and $c2$ if $\forall i,\ 0 \leqslant i \leqslant n$, $(c1 \in R(i) \implies c2 \notin R(i)) \wedge (c2 \in R(i) \implies c1 \notin R(i))$, i.e., for every step, if $c1$ ticks, $c2$ must not tick and vice versa.

The probabilistic extension of `causality` and `precedence` relations are defined based on the history of clocks.

Definition 7 (`Probabilistic Causality`). *The probabilistic* ***`causality`*** *relation between* $c1$ *and* $c2$ *(*$c1$ *is the cause and* $c2$ *is the effect), denoted* $c1 \preceq_p c2$*, is satisfied if the following condition holds:*

$$\mathcal{M} \models c1 \preceq_p c2 \iff Pr[c1 \preceq c2] \geqslant p$$

where $Pr[c1 \preceq c2] = \frac{1}{k} \sum_{j=1}^{k} \{R_j \models c1 \preceq c2\}$*, i.e., the ratio of* ***`runs`*** *satisfying the* ***`causality`*** *relation among the total number of* k ***`runs`****.*

A `run` R_j satisfies the `causality` relation on $c1$ and $c2$ if the condition holds: $\forall i$, $0 \leqslant i \leqslant n$, $H_R^{c1}(i) \geqslant H_R^{c2}(i)$. A tick of $c1$ satisfies `causality` relation if $c2$ does not occur prior to $c1$, i.e., the history of $c2$ is less than or equal to the history of $c1$ at the current step i.

The strict `causality`, called `precedence`, constrains that one clock must always tick faster than the other.

Definition 8 (`Probabilistic Precedence`). *The probabilistic* `precedence` *relation between* $c1$ *and* $c2$*, denoted* $c1 \prec_p c2$*, is satisfied if the following condition holds:*

$$\mathcal{M} \models c1 \prec_p c2 \iff Pr[c1 \prec c2] \geqslant p$$

where $Pr[c1 \prec c2] = \frac{1}{k} \sum_{j=1}^{k} \{R_j \models c1 \prec c2\}$ *is determined by the number of* `runs` *satisfying the* `precedence` *relation out of the* k `runs`*.*

A `run` R_j satisfies the `precedence` relation if the condition (expressed as (1)∧(2)) holds: $\forall i$, $0 \leqslant i \leqslant n$,

$$\underbrace{(H_R^{c1}(i) \geqslant H_R^{c2}(i))}_{(1)} \wedge \underbrace{(H_R^{c2}(i) = H_R^{c1}(i)) \implies (c2 \notin R(i))}_{(2)}$$

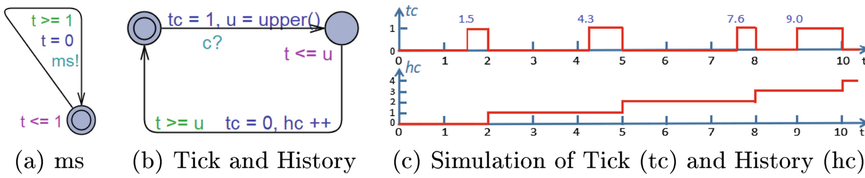
(1) The history of $c1$ is greater than or equal to the history of $c2$; (2) $c1$ and $c2$ must not be coincident, i.e., when the history of $c1$ and $c2$ are equal, $c2$ must not tick.

5 Translating CCSL and PrCCSL into UPPAAL-SMC

To formally verify the EAST-ADL timing constraints given in Sect. 3 using UPPAAL-SMC, we investigate how those constraints, specified in CCSL *expressions* and PrCCSL *relations*, can be translated into STA and probabilistic UPPAAL-SMC queries [12]. CCSL *expressions* construct new clocks and the *relations* between the new clocks are specified using PrCCSL. We first provide strategies that represent CCSL *expressions* as STA. We then present how the EAST-ADL timing constraints defined in PrCCSL can be translated into the corresponding STAs and UPPAAL-SMC queries based on the strategies.

5.1 Mapping CCSL to UPPAAL-SMC

We first describe how the universal clock (TimeUnit *ms*), tick and history of CCSL can be mapped to the corresponding STAs. Using the mapping, we then demonstrate that CCSL *expressions* can be modeled as STAs. The TimeUnit is implicitly represented as a single *step* of time progress in UPPAAL-SMC's *clock* [23]. The STA of TimeUnit (universal time defined as *ms*) consists of one location and one outgoing transition whereby the physical time and the duration



(a) ms (b) Tick and History (c) Simulation of Tick (tc) and History (hc)

Fig. 3. UPPAAL-SMC model of clock tick and history

of TimeUnit ms are represented by the *clock* variable t in Fig. 3(a). *clock* resets every time a transition is taken. The duration of TimeUnit is expressed by the invariant $t \leqslant 1$, and guard $t \geqslant 1$, i.e., a single step of the discrete time progress (tick) of universal time.

A clock c, considered as an event in UPPAAL-SMC, and its tick, i.e., an occurrence of the event, is represented by the synchronization channel $c!$. Since UPPAAL-SMC runs in chronometric semantics, in order to describe the discretized steps of `runs` (Rs), we consider if c ticks in the time range of $[i, i+1)$ ($i+1$ is excluded), c ticks at step i. The STA of tick and history is shown in Fig. 3(b). hc is the history of c, and tc indicates whether c ticks at the current step. A function *upper*() rounds the time instant (real number) up to the nearest greater integer. When c ticks via $c?$ at the current time step, tc is set to 1 prior to the time of the next step ($t < u$). hc is then increased by 1 (hc++) at the successive step (i.e., when $t = u$). For example, when c ticks at $time = 1.5$ (see Fig. 3(c)), *upper*() returns the value of 2 and tc becomes 1 during the time interval [1.5, 2), followed by hc being increased by 1 at $t = 2$.

Based on the mapping patterns of ms, tick and history, we present how `periodicOn`, `delayFor`, `infimum` and `supremum` *expressions* can be represented as UPPAAL-SMC models.

PeriodicOn: $c \triangleq$ `periodicOn` ms `period` q, where $\triangleq$ means "is defined as". `PeriodicOn` builds a new clock c based on ms and a *period* parameter q, i.e., c ticks at every q^{th} tick of ms. The STA of `periodicOn` is illustrated in Fig. 4(a). This STA initially stays in the *loop* location to detect q occurrences (ticks) of ms. The value x counts the number of ms ticks. When ms occurs ($ms?$), the STA takes the outgoing transition and increases x by 1. It "iterates" until ms ticks q times ($x == q$), then it activates the tick of c (via $c!$). At the successive step ($ms?$), it updates the history of c (hc++) and sets $x = 1$. The STA then returns to *loop* and repeats the calculation. This `periodicOn` STA can be used for the translation of EAST-ADL `Periodic` timing constraint (R1 in Fig. 1) into its UPPAAL-SMC model.

DelayFor: $c \triangleq c1$ `delayFor` d `on` $c2$. `delayFor` defines a new clock c based on $c1$ (*base* clock) and $c2$ (*reference* clock), i.e., each time $c1$ ticks, at the dth tick of $c2$, c ticks (each tick of c corresponds to a tick of $c1$). Kang et al. [23] and Suryadevara et al. [33] presented translation rules of `delayFor` into UPPAAL models. However, their approaches are not applicable in the case after $c1$ ticks, and $c1$ ticks again before the dth tick of $c2$ occurs. For example (see Fig. 2),

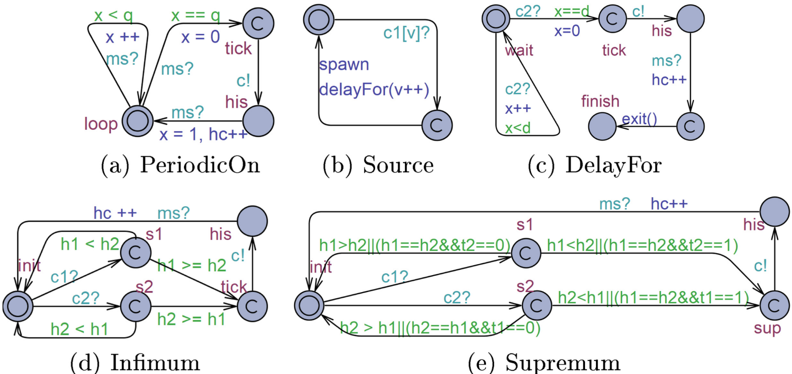


Fig. 4. STA of CCSL *expressions*

assume that d is 3. After the 1st tick of $c1$ (at step 0) happens, if $c1$ ticks again (at step 2) before the 3rd tick of $c2$ occurs (at step 4), the 2nd tick of $c1$ is discarded in their approaches. To alleviate the restriction, we utilize spawnable STA [12] as semantics denotation of `delayFor` expression and the STA of `delayFor` is shown in Fig. 4(c). As presented in Fig. 4(b), when the vth tick of $c1$ occurs ($c1[v]?$), its `delayFor` STA is spawned by `source` STA. The spawned STA stays in the *wait* location until $c2$ ticks d times. When $c2$ ticks d times ($x == d$), it transits to the *tick* location and triggers c ($c!$). At the next step ($ms?$), the STA increases hc by 1 and moves to $finish$ location and then becomes inactive, i.e., calculation of the vth tick of c is completed. This `delayFor` STA can be utilized to construct the UPPAAL-SMC models of EAST-ADL timing requirements R3 – R7 in Sect. 3.

Given two clocks $c1$ and $c2$, their `infimum` (resp. `supremum`) is informally defined as the slowest (resp. fastest) clock faster (resp. slower) than both $c1$ and $c2$. `infimum` and `supremum` are useful in order to group events occurring at the same time and decide which one occurs first and which one occurs last. The representative STAs for both *expressions* are utilized for the translation of EAST-ADL `Synchronization` timing constraint (R5 in Sect. 3) into the UPPAAL-SMC model.

Infimum creates a new clock c, which is the slowest clock faster than $c1$ and $c2$. The STA of `infimum` is illustrated in Fig. 4(d). When $c1$ ($c2$) ticks via $c1?$ ($c2?$), the STA transits to the *s1* (*s2*) location and compares the history of the two clocks ($h1$ and $h2$) to check whether the current ticking clock $c1$ ($c2$) is faster than $c2$ ($c1$). If so, i.e., the condition "$h1 \geqslant h2$ ($h2 \geqslant h1$)" holds, the STA takes a transition to the *tick* location and activates the tick of c ($c!$). After updating the history (hc++), it returns to the *init* location and repeats the calculation.

Supremum builds a new clock c, which is the fastest clock slower than $c1$ and $c2$. It states that if $c1$ ticks at the current step and $c1$ is slower than $c2$, then c ticks. The STA of `supremum` is shown in Fig. 4(e). When $c1$ ($c2$) ticks via $c1?$ ($c2?$), the STA transits to the *s1* (*s2*) location and compares the history of the two clocks and decides whether $c1$ ($c2$) is slower than $c2$ ($c1$). If $c1$ ($c2$) ticks

slower than $c2$ ($c1$), i.e., $h1 < h2$ ($h2 < h1$), or $c1$ and $c2$ tick at the same rate, i.e., "$h1 == h2$ && $t_2 == 1$ ($h1 == h2$ && $t_1 == 1$)" holds, the tick of c is triggered. The STA then updates the history of c and goes back to *init* and repeats the process.

5.2 Representation of PrCCSL in UPPAAL-SMC

In this section, the translation of East-adl timing constraints specified in PrCcsl into STA and *Hypothesis Testing* query (refer to Sect. 2) is provided from the view point of the analysis engine Uppaal-SMC.

Recall the definition of PrCcsl in Sect. 4. The probability of a *relation* being satisfied is interpreted as a ratio of runs that satisfies the *relation* among all runs. It is specified as *Hypothesis Testing* queries in Uppaal-SMC, H_0: $\frac{m}{k} \geqslant P$ against H_1: $\frac{m}{k} < P$, where m is the number of runs satisfying the given *relation* out of all k runs. k is decided by strength parameters α (the probability of false positives, i.e., accepting H_1 when H_0 holds) and β (probability of false negatives, i.e., accepting H_0 when H_1 holds), respectively [10].

Based on the mapping patterns of tick and history in Sect. 5.1, the probabilistic extension of `exclusion`, `causality` and `precedence` relations are expressed as *Hypothesis Testing* queries straightforwardly.

Probabilistic Exclusion is employed to specify East-adl `Exclusion` timing constraint, $turnLeft\ \#_p\ rightOn$ (Spec. R8 in Fig. 1). It states that the two events, *turnLeft* and *rightOn* (the vehicle is turning left and right), must be exclusive. The ticks of $turnLeft$ and $rightOn$ events are modeled using the STA in Fig. 3(b). Based on the definition of `probabilistic exclusion` (Sect. 4), R8 is expressed in *Hypothesis Testing* query: $Pr[bound]\ ([\ ]((t_{turnLeft} \implies \neg\ t_{rightOn}) \wedge (t_{rightOn} \implies \neg\ t_{turnLeft}))) \geqslant P$, where $t_{turnLeft}$ and $t_{rightOn}$ indicate the ticks of $turnLeft$ and $rightOn$, respectively. *bound* is the time bound of simulation, in our setting $bound = 3000$.

Probabilistic Causality is used to specify East-adl `Synchronization` timing constraint, $sup \preceq_p \{inf$ `delayFor` 40 `on` $ms\}$ (Spec. R5 in Fig. 1), where *sup* (*inf*) is the fastest (slowest) event slower (faster) than five input events, *speed*, *signType*, *direct*, *gear* and *torque*. Let `SUP` and `INF` denote the `supremum` and `infimum` operator, i.e., `SUP`($c1$, $c2$) (resp. `INF`($c1$, $c2$)) returns the `supremum` (resp. `infimum`) of clock $c1$ and $c2$. *sup* and *inf* can now be expressed with the nested operators (where $\triangleq$ means "is defined as"):

$$sup \triangleq \texttt{SUP}(speed,\ \texttt{SUP}(\texttt{SUP}(signType,\ direct),\ \texttt{SUP}(gear,\ torque)))$$

$$inf \triangleq \texttt{INF}(speed,\ \texttt{INF}(\texttt{INF}(signType,\ direct),\ \texttt{INF}(gear,\ torque)))$$

For the translation of *sup* (*inf*) into Uppaal-SMC model, we employ the STA of `supremum` (resp. `infimum`) (Fig. 4(d) and (e)) for each `SUP` (`INF`) operator. A new clock *dinf* is generated by delaying *inf* for 40 ticks of *ms*: $dinf \triangleq \{inf$ `delayFor` 40 `on` $ms\}$. The Uppaal-SMC model of *dinf* is achieved by adapting the spawnable *DelayFor* STA (Fig. 4). Based on the `probabilistic`

causality definition, R5 is interpreted as: $Pr[\leqslant bound]([\,]\, h_{sup} \geqslant h_{dinf}) \geqslant P$, where h_{sup} and h_{dinf} are the history of *sup* and *dinf* respectively. Similarly, Execution (R3) and Comparison (R7) timing constraints specified in probabilistic causality using delayFor can be translated into *Hypothesis Testing* queries. For further details, refer to the technical report [20].

Probabilistic Precedence is utilized to specify East-adl End-to-End timing constraint (R4). It states that the time duration between the *source* event *signIn* (input signal on the *signType* port of Controller) and the *target* event *spOut* (output signal on the *speed* port of VehicleDynamic) must be within a time bound of [150, 250], and that is specified as Uppaal-SMC queries (1) and (2):

$$\{signIn \ \texttt{delayFor}\ 150\ \texttt{on}\ ms\} \prec_p spOut \tag{1}$$

$$spOut \prec_p \{signIn \ \texttt{delayFor}\ 250\ \texttt{on}\ ms\} \tag{2}$$

Two clocks, *lower* and *upper*, are defined by delaying *signIn* for 150 and 250 ticks of *ms* respectively: $lower \triangleq \{signIn \ \texttt{delayFor}\ 150\ \texttt{on}\ ms\}$, and $upper \triangleq \{signIn \ \texttt{delayFor}\ 250\ \texttt{on}\ ms\}$. The corresponding Uppaal-SMC models of *lower* and *upper* are constructed based on the delayFor STA (shown in Fig. 4). Finally, R4 specified in PrCcsl is expressed as Uppaal-SMC queries (3) and (4), where $h_{\texttt{lower}}$, $h_{\texttt{upper}}$ and $h_{\texttt{spOut}}$ are the history of *lower*, *upper* and *spOut*. $t_{\texttt{spOut}}$ and $t_{\texttt{upper}}$ represent the tick of *upper* and *spOut* respectively:

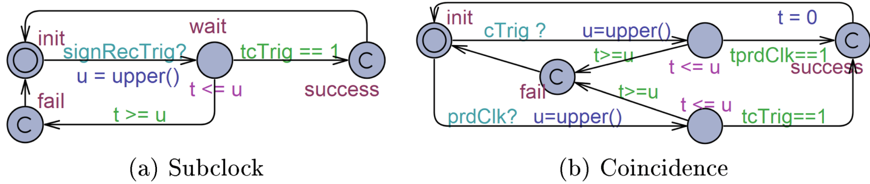
$$Pr[\leqslant bound]([\,]h_{\texttt{lower}} \geqslant h_{\texttt{spOut}} \wedge ((h_{\texttt{lower}} == h_{\texttt{spOut}}) \implies t_{\texttt{spOut}} == 0)) \geqslant P \tag{3}$$

$$Pr[\leqslant bound]([\,]h_{\texttt{spOut}} \geqslant h_{\texttt{upper}} \wedge ((h_{\texttt{spOut}} == h_{\texttt{upper}}) \implies t_{\texttt{upper}} == 0)) \geqslant P \tag{4}$$

Similarly, East-adl Sporadic timing constraint (R6) specified in probabilistic precedence can be translated into *Hypothesis Testing* query [20].

In the case of properties specified in either probabilistic subclock or probabilistic coincidence, such properties cannot be directly expressed as Uppaal-SMC queries. Therefore, we construct observer STA that capture the semantics of standard subclock and coincidence *relations*. The observer STA are composed to the system STA (namely a network STA, NSTA) in parallel. Then, the probabilistic analysis is performed over the NSTA which enables us to verify the East-adl timing constraints specified in probabilistic subclock and probabilistic coincidence of the entire system using Uppaal-SMC. Further details are given below.

Probabilistic Subclock is employed to specify East-adl Periodic timing constraint, given as $signRecTrig \subseteq_p cTrig$ (Spec. R2 in Fig. 1). The standard subclock *relation* states that superclock must tick at the same step where subclock ticks. Its corresponding STA is shown in Fig. 5(a). When *signRevTrig* ticks (*signRecTrig*?), the STA transits to the *wait* location and detects the occurrence of *cTrig* until the time point of the subsequent step (u). If *cTrig* occurs prior to the next step ($tcTrig == 1$), the STA moves to the *success* location,



(a) Subclock (b) Coincidence

Fig. 5. Observer STA of `Subclock` and `Coincidence`

i.e., the `subclock` *relation* is satisfied at the current step. Otherwise, it transits to the $fail$ location. R2 specified in `probabilistic subclock` is expressed as: $Pr[bound]([\,]\neg\ Subclock.fail) \geqslant P$. UPPAAL-SMC analyzes if the $fail$ location is never reachable from the system NSTA, and whether the probability of R2 being satisfied is greater than or equal to P.

Probabilistic Coincidence is adapted to specify EAST-ADL `Periodic` timing constraint, given as $cTrig \equiv_p$ {`periodicOn` ms `period` 50} (Spec. R1 in Fig. 1). To express R1 in UPPAAL-SMC, first, a periodic clock $prdClk$ ticking every 50th tick of ms is defined: $prdClk \triangleq$ `periodicOn` ms `period` 50. The corresponding UPPAAL-SMC model of $prdClk$ is generated based on the `periodicOn` STA shown in Fig. 4(a) by setting q as 50. Then, we check if $cTrig$ and $prdClk$ are coincident by employing the `coincidence` STA shown in Fig. 5(b). When $cTrig$ ($prdClk$) ticks via $cTrig$? ($prdClk$?), the STA checks if the other clock, $prdClk$ ($cTrig$), ticks prior to the next step, i.e., whether $tprdClk == 1$ ($tcTrig == 1$) holds or not when $t \leqslant u$. The STA then transits to either the $success$ or $fail$ location based on the judgement. R1 specified in `probabilistic coincidence` is expressed as: $Pr[bound]([\,]\neg\ Coincidence.fail) \geqslant P$. UPPAAL-SMC analyzes if the probability of R1 being satisfied is greater than or equal to P.

6 Experiments: Verification and Validation

We have formally analyzed over 30 properties (associated with timing constraints) of the system including *deadlock freedom* [20]. A list of selected properties (Sect. 3) are verified using UPPAAL-SMC and the results are listed in Table.1. Five types of UPPAAL-SMC queries are employed to specify R1–R8, *Hypothesis Testing* (HT), *Probability Estimation* (PE), *Probability Comparison* (PC), *Expected Value* (EV) and *Simulations* (SI).

1. *Hypothesis Testing:* All properties are established as valid with 95% level of confidence; 2. *Probability Estimation:* The probability of each property being satisfied is computed and its approximate interval is given as [0.902, 1]; 3. *Expected Value:* The expected values of time durations of timing constraints (R1 – R7) are evaluated. For example, during the analysis of R1, the time interval between two consecutive triggerings of the `Camera` is evaluated as 50 and that validates R1. Furthermore, UPPAAL-SMC evaluates the expected maximum duration bound of `End-to-End` timing constraint by checking R4 and generates the frequency

Table 1. Verification results in UPPAAL-SMC

R	Q	Expression	Result	Time	Mem	CPU
R1	HT	$\Pr[\leqslant 3000]([\,]\ \neg Coin.fail) \geqslant 0.95$	Valid	48.7	32.7	31.3
	PE	$\Pr[\leqslant 3000]([\,]\ \neg Coin.fail)$	[0.902, 1]	12.6	35.6	29.8
	EV	$E[\leqslant 3000;\ 500]([\,]\ max : cam.t)$	50 ± 0	83.3	33.3	31.7
	SI	simulate 500 $[\leqslant 3000](camtrig,\ p1trig)$	Valid	80.9	32.9	32.5
R2	HT	$\Pr[\leqslant 3000]([\,]\ \neg Sub.fail) \geqslant 0.95$	Valid	48.9	32.9	29.3
	PE	$\Pr[\leqslant 3000]([\,]\ \neg Sub.fail)$	[0.902, 1]	12.3	35.5	30.4
	EV	$E[\leqslant 3000;\ 500]([\,]\ max : sf.t)$	200 ± 0	80.6	32.5	32.2
	SI	simulate 500 $[\leqslant 3000](strig,\ p2trig)$	Valid	85.5	33.1	32.3
R3	HT	$\Pr[\leqslant 3000]([\,]\ h_{\mathtt{SU}} \leqslant h_{\mathtt{S}}) \geqslant 0.95$	Valid	76.5	40.4	32.3
	PE	$\Pr[\leqslant 3000]([\,]\ h_{\mathtt{SU}} \leqslant h_{\mathtt{S}})$	[0.902, 1]	18.1	40.3	30.8
	HT	$\Pr[\leqslant 3000]([\,]\ h_{\mathtt{S}} \leqslant h_{\mathtt{SL}}) \geqslant 0.95$	Valid	77.6	37.7	31.7
	PE	$\Pr[\leqslant 3000]([\,]\ h_{\mathtt{S}} \leqslant h_{\mathtt{SL}})$	[0.902, 1]	16.5	40.0	31.5
	PC	$\Pr[\leqslant 3000]\ ([\,]\ SR.exec \implies (SR.t \geqslant 100 \wedge SR.t \leqslant 125)) \geqslant$ $\Pr[\leqslant 3000]\ ([\,]\ SR.exec \implies (SR.t \geqslant 125 \wedge SR.t \leqslant 150))$	$\geqslant$1.1	8.3	31.7	32.3
	EV	$E[\leqslant 3000;\ 500]([\,]\ max : checkexe.t)$	147.2 ± 0.7	82.8	32.6	30.4
	SI	simulate 500 $[\leqslant 3000](h_{\mathtt{SU}}, h_{\mathtt{S}}, h_{\mathtt{SL}})$	Valid	86.9	33.2	33.4
R4	HT	$\Pr[\leqslant 3000]([\,]h_{\mathtt{lower}} \geqslant h_{\mathtt{spOut}} \wedge ((h_{\mathtt{lower}} == h_{\mathtt{spOut}}) \implies t_{\mathtt{spOut}} == 0)) \geqslant 0.95$	Valid	54.2	32.9	31.4
	PE	$\Pr[\leqslant 3000]([\,]h_{\mathtt{lower}} \geqslant h_{\mathtt{spOut}} \wedge ((h_{\mathtt{lower}} == h_{\mathtt{spOut}}) \implies \neg t_{\mathtt{spOut}}))$	[0.902, 1]	13.1	35.3	29.4
	HT	$\Pr[\leqslant 3000]([\,]h_{\mathtt{spOut}} \geqslant h_{\mathtt{upper}} \wedge ((h_{\mathtt{spOut}} == h_{\mathtt{upper}}) \implies t_{\mathtt{upper}} == 0)) \geqslant 0.95$	Valid	1.3 h	32.2	32.6
	PE	$\Pr[\leqslant 3000]([\,]h_{\mathtt{spOut}} \geqslant h_{\mathtt{upper}} \wedge ((h_{\mathtt{spOut}} == h_{\mathtt{upper}}) \implies \neg t_{\mathtt{upper}}))$	[0.902, 1]	19.8	34.1	32.0
	EV	$E[\leqslant 3000;\ 500]([\,]\ max : checke2e.t)$	229.7 ± 0.9	83.3	32.5	30.6
	SI	simulate 500 $[\leqslant 3000](h_{\mathtt{CU}}, h_{\mathtt{VD}}, h_{\mathtt{CL}}, t_{\mathtt{CU}}, t_{\mathtt{VD}})$	Valid	89.8	32.9	30.2
R5	HT	$\Pr[\leqslant 3000]([\,]\ h_{dinf} \geqslant h_{sup}) \geqslant 0.95$	Valid	53.9	32.7	31.9
	PE	$\Pr[\leqslant 3000]([\,]\ h_{dinf} \geqslant h_{sup})$	[0.902, 1]	13.7	35.5	30.4
	EV	$E[\leqslant 3000;\ 500]([\,]\ max : checksync.t)$	30.6 ± 0.21	72.4	32.6	31.6
	SI	simulate 500 $[\leqslant 3000](h_{dinf}, h_{sup})$	Valid	86.8	32.6	32.0
R6	HT	$\Pr[\leqslant 3000]([\,]\ hv \leqslant ho \wedge ((hv == ho) \implies t_{\mathtt{va}} == 0)) \geqslant 0.95$	Valid	3h	33.1	30.0
	PE	$\Pr[\leqslant 3000]([\,]\ hv \leqslant ho \wedge ((hv == ho) \implies t_{\mathtt{va}} == 0))$	[0.902, 1]	45.4	33.1	29.4
	EV	$E[\leqslant 3000;\ 500]([\,]\ max : obs.t)$	667 ± 79	80.8	29.7	31.7
	SI	simulate 500 $[\leqslant 3000](hv, ho, v))$	Valid	88.6	29.5	31.0
R7	HT	$\Pr[\leqslant 3000]([\,]$ $(ex_{\mathtt{con}} == wcet_{\mathtt{con}} \wedge ex_{\mathtt{vd}} == wcet_{\mathtt{vd}}) \implies (h_{\mathtt{cu}} \geqslant h_{\mathtt{com}})) \geqslant$ 0.95	Valid	57.4	36.7	28.4
	PE	$\Pr[\leqslant 3000]([\,]$ $(ex_{\mathtt{con}} == wcet_{\mathtt{con}} \wedge ex_{\mathtt{vd}} == wcet_{\mathtt{vd}}) \implies (h_{\mathtt{cu}} \geqslant h_{\mathtt{com}}))$	[0.902, 1]	14.7	35.5	26.7
	EV	$E[\leqslant 3000;\ 500]([\,]\ max : control.t)$	146.7 ± 0.28	74.9	29.4	32.7
	EV	$E[\leqslant 3000;\ 500]([\,]\ max : vd.t)$	96.6 ± 0.27	74.2	29.4	31.4
	SI	simulate 500 $[\leqslant 3000](h_{\mathtt{cu}}, h_{\mathtt{com}}))$	Valid	86.6	29.5	32.5
R8	HT	$\Pr[\leqslant 3000]([\,]\ \neg(t_{\mathtt{Right}} == 1\ \wedge\ t_{\mathtt{Left}} == 1)) \geqslant 0.95$	Valid	57.4	36.7	28.4
	PE	$\Pr[\leqslant 3000]([\,]\ \neg(t_{\mathtt{Right}} == 1\ \wedge\ t_{\mathtt{Left}} == 1))$	[0.902, 1]	14.7	35.5	26.7
	SI	simulate 500 $[\leqslant 3000](t_{\mathtt{Right}}, t_{\mathtt{Left}})$	Valid	85.5	29.6	32.6

histogram of the expected bound (see Fig. 7). It illustrates that the expected bound is always less than 250 ms and 90% of the duration is within the range of [207, 249]; 4. *Probability Comparison:* is applied to confirm that the probability of `SignRecognition` f_p completing its execution within [100, 125] ms is greater than the probability of completion within [125, 150] ms (R3). The query results in a comparison probability ratio greater than or equal to 1.1, i.e., the execution time of `SignRecognition` f_p is most likely less than 125 ms. 5. *Simulation*: The

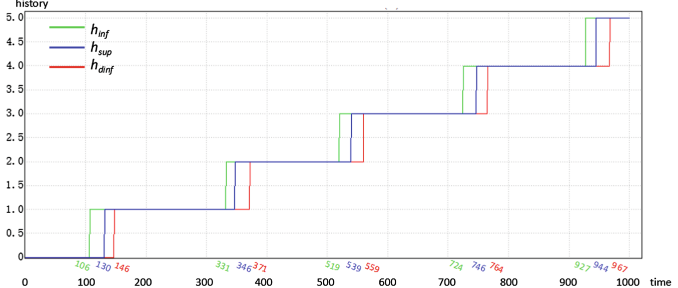


Fig. 6. Simulation result of R6. (Color figure online)

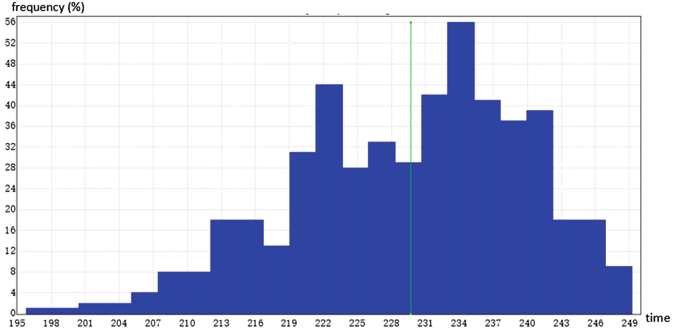


Fig. 7. Frequency histogram of `End-to-End` timing constraint (R4)

simulation result of `Synchronization` timing constraint (R5) is demonstrated in Fig. 6. h_{inf}, h_{sup} and h_{dinf} are history of inf, sup and $dinf$ respectively. Recall Spec. R5 (see Fig. 1), the `causality` *relation* between $dinf$ and sup is satisfied. As the simulation of R6 shows (Fig. 6), the rising edge of h_{sup} (in blue) always occurs prior to h_{dinf} (in red). It indicates that sup always runs faster than $dinf$, thus the `causality` *relation* is validated.

7 Related Work

In the context of East-adl, efforts on the integration of East-adl and formal techniques based on timing constraints were investigated in several works [15,17,24,31], which are however, limited to the executional aspects of system functions without addressing stochastic behaviors. Kang [23] and Suryadevara [33,34] defined the execution semantics of both the controller and the environment of industrial systems in Ccsl which are also given as mapping to Uppaal models amenable to model checking. In contrast to our current work, those approaches lack precise stochastic annotations specifying continuous dynamics in particular regarding different clock rates during execution. Ling [35] transformed a subset of Ccsl constraints to PROMELA models to perform formal

verification using SPIN. Zhang [36] transformed Ccsl into first order logics that are verifiable using SMT solver. However, their works are limited to functional properties, and no timing constraints are addressed. Though, Kang et al. [16,19] and Marinescu et al. [28] present both simulation and model checking approaches of Simulink and Uppaal-SMC on East-adl models, neither formal specification nor verification of extended East-adl timing constraints with probability were conducted. Our approach is a first application on the integration of East-adl and formal V&V techniques based on probabilistic extension of East-adl/Tadl2 constraints using PrCcsl and Uppaal-SMC. An earlier study [18,21,22] defined a probabilistic extension of East-adl timing constraints and presented model checking approaches on East-adl models, which inspires our current work. Specifically, the techniques provided in this paper define new operators of Ccsl with stochastic extensions (PrCcsl) and verify the extended East-adl timing constraints of CPS (specified in PrCcsl) with statistical model checking. Du et al. [13] proposed the use of Ccsl with probabilistic logical clocks to enable stochastic analysis of hybrid systems by limiting the possible solutions of clock ticks. Whereas, our work is based on the probabilistic extension of East-adl timing constraints with a focus on probabilistic verification of the extended constraints, particularly, in the context of WH.

8 Conclusion

We present an approach to perform probabilistic verification on East-adl timing constraints of automotive systems based on WH at the early design phase: 1. Probabilistic extension of Ccsl, called PrCcsl, is defined and the East-adl/Tadl2 timing constraints with stochastic properties are specified in PrCcsl; 2. The semantics of the extended constraints in PrCcsl is translated into verifiable Uppaal-SMC models for formal verification; 3. A set of mapping rules is proposed to facilitate guarantee of translation. Our approach is demonstrated on an autonomous traffic sign recognition vehicle (AV) case study. Although, we have shown that defining and translating a subset of Ccsl with probabilistic extension into Uppaal-SMC models is sufficient to verify East-adl timing constraints, as ongoing work, advanced techniques covering a full set of Ccsl constraints are further studied. Despite the fact that Uppaal-SMC supports probabilistic analysis of the timing constraints of AV, the computational cost of verification in terms of time is rather expensive. Thus, we continue to investigate complexity-reducing design/mapping patterns for CPS to improve effectiveness and scalability of system design and verification.

Acknowledgment. This work is supported by the NSFC, EASY Project: 46000-41030005.

References

1. Automotive Open System Architecture. https://www.autosar.org/
2. UPPAAL-SMC. http://people.cs.aau.dk/~adavid/smc/
3. IEC 61508: Functional Safety of Electrical Electronic Programmable Electronic Safety Related Systems. International Organization for Standardization, Geneva (2010)
4. ISO 26262–6: Road Vehicles Functional Safety Part 6. Product Development at the Software Level. International Organization for Standardization, Geneva (2011)
5. MAENAD (2011). http://www.maenad.eu/
6. André, C.: Syntax and semantics of the clock constraint specification language (CCSL). Ph.D. thesis, INRIA (2009)
7. André, C., Mallet, F.: Clock constraints in UML/MARTE CCSL. HAL - INRIA (2008)
8. Bernat, G., Burns, A., Llamosi, A.: Weakly hard real-time systems. Trans. Comput. **50**(4), 308–321 (2001)
9. Blom, H., et al.: TIMMO-2-USE timing model, tools, algorithms, languages, methodology, use cases. Technical report, TIMMO-2-USE (2012)
10. Bulychev, P., et al.: UPPAAL-SMC: statistical model checking for priced timed automata. In: QAPL, pp. 1–16. EPTCS (2012)
11. David, A., et al.: Statistical model checking for stochastic hybrid systems. In: HSB, pp. 122–136. EPTCS (2012)
12. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B.: UPPAAL-SMC tutorial. STTT **17**(4), 397–415 (2015)
13. Du, D., Huang, P., Jiang, K., Mallet, F., Yang, M.: MARTE/pCCSL: modeling and refining stochastic behaviors of CPSs with probabilistic logical clocks. In: Kouchnarenko, O., Khosravi, R. (eds.) FACS 2016. LNCS, vol. 10231, pp. 111–133. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57666-4_8
14. EAST-ADL Consortium: EAST-ADL domain model specification v2.1.9. Technical report, MAENAD European Project (2011)
15. Goknil, A., Suryadevara, J., Peraldi-Frati, M.-A., Mallet, F.: Analysis support for TADL2 timing constraints on EAST-ADL models. In: Drira, K. (ed.) ECSA 2013. LNCS, vol. 7957, pp. 89–105. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39031-9_8
16. Kang, E.Y., Chen, J., Ke, L., Chen, S.: Statistical analysis of energy-aware real-time automotive systems in EAST-ADL/Stateflow. In: ICIEA, pp. 1328–1333. IEEE (2016)
17. Kang, E.Y., Enoiu, E.P., Marinescu, R., Seceleanu, C., Schobbens, P.Y., Pettersson, P.: A methodology for formal analysis and verification of EAST-ADL models. Reliabil. Eng. Syst. Saf. **120**(12), 127–138 (2013)
18. Kang, E.Y., Huang, L., Mu, D.: Formal verification of energy and timed requirements for a cooperative automotive system. In: SAC, pp. 1492–1499. ACM (2018)
19. Kang, E.Y., Ke, L., Hua, M.Z., Wang, Y.X.: Verifying automotive systems in EAST-ADL/Stateflow using UPPAAL. In: APSEC, pp. 143–150. IEEE (2015)
20. Kang, E.Y., Mu, D., Huang, L.: Probabilistic analysis of weakly-hard real-time systems. Technical report, SYSU (2018). https://sites.google.com/site/kangeu/home/publications
21. Kang, E.Y., Mu, D., Huang, L., Lan, Q.: Model-based analysis of timing and energy constraints in an autonomous vehicle system. In: QRS, pp. 525–532. IEEE (2017)

22. Kang, E.Y., Mu, D., Huang, L., Lan, Q.: Verification and validation of a cyber-physical system in the automotive domain. In: QRS, pp. 326–333. IEEE (2017)
23. Kang, E.Y., Schobbens, P.Y.: Schedulability analysis support for automotive systems: from requirement to implementation. In: SAC, pp. 1080–1085. ACM (2014)
24. Kang, E.-Y., Schobbens, P.-Y., Pettersson, P.: Verifying functional behaviors of automotive products in EAST-ADL2 using UPPAAL-PORT. In: Flammini, F., Bologna, S., Vittorini, V. (eds.) SAFECOMP 2011. LNCS, vol. 6894, pp. 243–256. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24270-0_18
25. Legay, A., Viswanathan, M.: Statistical model checking: challenges and perspectives. STTT **17**(4), 369–376 (2015)
26. Mallet, F., Peraldi-Frati, M.A., Andre, C.: MARTE CCSL to execute EAST-ADL timing requirements. In: ISORC, pp. 249–253. IEEE (2009)
27. Mallet, F., De Simone, R.: Correctness issues on MARTE/CCSL constraints. Sci. Comput. Program. **106**, 78–92 (2015)
28. Marinescu, R., Kaijser, H., Mikučionis, M., Seceleanu, C., Lönn, H., David, A.: Analyzing industrial architectural models by simulation and model-checking. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2014. CCIS, vol. 476, pp. 189–205. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17581-2_13
29. Nicolau, G.B.: Specification and analysis of weakly hard real-time systems. Trans. Comput. 308–321 (1988)
30. Object Management Group: UML profile for MARTE: Modeling and analysis of real-time embedded systems (2015)
31. Qureshi, T.N., Chen, D.J., Persson, M., Törngren, M.: Towards the integration of UPPAAL for formal verification of EAST-ADL timing constraint specification. In: TiMoBD Workshop (2011)
32. Simulink and Stateflow. https://www.mathworks.com/products.html
33. Suryadevara, J.: Validating EAST-ADL timing constraints using UPPAAL. In: SEAA, pp. 268–275. IEEE (2013)
34. Suryadevara, J., Seceleanu, C., Mallet, F., Pettersson, P.: Verifying MARTE/CCSL mode behaviors using UPPAAL. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM 2013. LNCS, vol. 8137, pp. 1–15. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40561-7_1
35. Yin, L., Mallet, F., Liu, J.: Verification of MARTE/CCSL time requirements in PROMELA/SPIN. In: ICECCS, pp. 65–74. IEEE (2011)
36. Zhang, M., Ying, Y.: Towards SMT-based LTL model checking of clock constraint specification language for real-time and embedded systems. ACM SIGPLAN Not. **52**(4), 61–70 (2017)

Facilitating the Implementation of Distributed Systems with Heterogeneous Interactions

Salwa Kobeissi[1], Adnan Utayim[2], Mohamad Jaber[2](✉), and Yliès Falcone[3]

[1] University of Strasbourg, Inria, ICube Laboratory, Strasbourg, France
salwa.kobeissi@inria.fr
[2] Computer Science Department, American University of Beirut, Beirut, Lebanon
{mmu00,mj54}@aub.edu.lb
[3] Univ. Grenoble Alpes, Inria, LIG, 38000 Grenoble, France
ylies.falcone@univ-grenoble-alpes.fr

Abstract. We introduce HDBIP an extension of the Behavior Interaction Priority (BIP) framework. BIP is a component-based framework with a rigorous operational semantics and high-level and expressive interaction model. HDBIP extends BIP interaction model by allowing heterogeneous interactions targeting distributed systems. HDBIP allows both multiparty and direct send/receive interactions that can be directly mapped to an underlying communication library. Then, we present a correct and efficient code generation from HDBIP to C++ implementation using Message Passing Interface (MPI). We present a non-trivial case study showing the effectiveness of HDBIP.

1 Introduction

Developing correct and reliable distributed systems is challenging mainly because of the complex structures of the interactions between distributed processes. On the one hand, the use of abstract interaction models may simplify the development process but may deteriorate the performance of the generated implementation. On the other hand, the use of low-level primitives makes modeling error prone and time consuming. Although different frameworks [3,15] exist to model interactions between distributed processes, building correct, reliable and scalable distributed systems is still challenging and a hardly predictive task.

In this paper, we introduce HDBIP an extension of the Behavior, Interaction, and Priority (BIP) framework. BIP is a component-based framework used to model heterogeneous and complex systems. BIP has an expressive interaction model [5] that handles synchronization and communication between processes/components. Using only multiparty interactions simplifies the modeling of distributed barriers with local non-determinism, by automatically generating controllers to handle conflicts [6]. Nonetheless, restricting the language to only multiparty interactions affects the performance of the distributed implementations for instance to model a simple asynchronous send/receive primitive.

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 255–274, 2018.
https://doi.org/10.1007/978-3-319-98938-9_15

In that case, the implementation requires an explicit buffer component/process. As such, this allows the creation of extra processes that are not needed. This extra buffer is practically duplicated as system buffers are usually provided by the low-level communication libraries (e.g., MPI). Moreover, it is required to use multiparty interactions to connect the send primitives and the receive primitives with the explicit buffers. As such, those connections may introduce conflicts between themselves and between multiparty interactions, which may drastically affect the performance of the distributed implementation.

This paper introduces `HDBIP`, which allows the modeling of both multiparty and asynchronous send receive interactions in an elegant way. Moreover, we provide an efficient code generation that allows by-construction to directly execute the send receive interactions with no need to create the extra buffers and instead use the system buffers. We show the effectiveness of `HDBIP` on distributed two-phase commit protocol. We mainly compare with respect to BIP the execution time and the lines of code needed.

The remainder of this paper is structured as follows. Section 2 presents the existing BIP framework. Section 3 introduces `HDBIP`, an extension of BIP. Section 4 defines how it is possible to generate efficient implementations from `HDBIP` along with the arguments supporting correctness of the generated implementation. In Sect. 5, we evaluate the performance of `HDBIP` by comparing it to BIP. Section 6 presents related work. Finally, Sect. 7 draws some conclusions and presents future work.

2 Behavior Interaction Protocol (BIP) Framework

The Behavior Interaction Priority (BIP) framework [3] offers high-level synchronization primitives that simplify system development and allow for the generation of both centralized and distributed implementations from high-level models. It consists of three layers: Behavior, Interaction and Priority. *Behavior* is expressed by Labeled Transition Systems (LTS) describing atomic components extended with data and C functions. Moreover, transitions of atomic components are labeled with ports that are exported for communication/synchronization with other components. *Interaction* models the synchronization and communication between ports of atomic components. *Priority* specifies scheduling constraints on interactions.

2.1 Atomic Components

Let us consider a set of local variables X.

Definition 1 (Port). *A port is a tuple* $\langle p, X_p \rangle$ *where* p *is an identifier and* $X_p \subseteq X$ *is a set of exported local variables. A port is referred to by its identifier.*

Definition 2 (Atomic component - Syntax). *An atomic component is a tuple* $\left\langle P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T} \right\rangle$*, such that:*

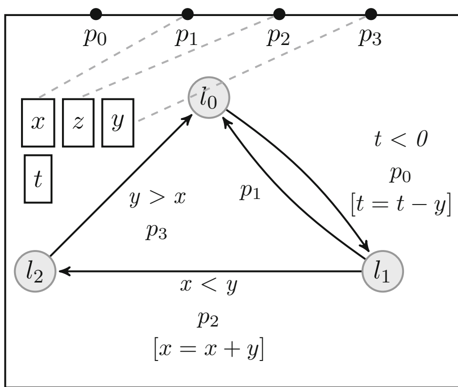


Fig. 1. Atomic component in BIP

- $\langle P, L, T \rangle$ *is an LTS over a set of ports* P, L *is a set of control locations, and* $T \subseteq L \times P \times L$ *is a set of transitions;*
- X *is a finite set of variables;*
- *Every transition* $\tau \in T$ *has a guard* g_τ *(a predicate over* X*), and a function* $f_\tau \in \{x := f^x(X) \mid x \in X\}^*$*, triggered by this transition, that updates the values of variables in* X*.*

A transition $\tau = \langle l, p, l' \rangle \in T$, where l (resp. l') is the source (resp. destination) of τ. p is the label of τ used as an interface to synchronize with other components. Moreover, a transition can be augmented with a guard g_τ and a function f_τ, thus defined as $\tau = \langle l, p, g_\tau, f_\tau, l' \rangle$. The port attached to a transition is said to be *enabled* only if the guard of the transition g_τ holds.

Definition 3 (Atomic component - semantics). *The semantics of atomic component* $\langle P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T} \rangle$ *is the LTS* $\langle Q, P, T_0 \rangle$*, where:*

- $Q = L \times [X \to \text{Data}] \times (P \cup \{\texttt{null}\})$*;*
- $T_0 = \{\langle \langle l, v, p \rangle, p'(v_{p'}), \langle l', v', p' \rangle \rangle \in Q \times P \times Q \mid \exists \tau = \langle l, p', l' \rangle \in T : g_\tau(v) \wedge v' = f_\tau(v/v_{p'})\}$*, where* $v_{p'} \in [X_{p'} \to \text{Data}]$*.*

A configuration/state of an atomic component is a triple $\langle l, v, p \rangle \in Q$ where $l \in L, v \in [X \to \text{Data}]$ is a valuation of variables in X, and $p \in P$ is the port of the last-executed transition (or `null` otherwise, i.e., in case of the initial configuration). The evolution $\langle l, v, p \rangle \stackrel{p'(v_{p'})}{\to} \langle l', v', p' \rangle$, where $v_{p'}$ is a valuation of the variables in $X_{p'}$, is possible if there exists a transition $\langle l, p', g_\tau, f_\tau, l' \rangle$, s.t. p' is enabled or $g_\tau(v) = \texttt{true}$. Valuation v is modified to $v' = f_\tau(v/v_{p'})$.

We use the dot notation to denote the elements of an atomic component B. For instance, we refer to its set of ports as $B.P$, its set of locations as $B.L$ and its set of local variables as $B.X$.

Figure 1 depicts an atomic component B. B has four ports p_0, p_1, p_2 and p_3 and four local variables x, y, z and t. Port p_1 exports variable x, p_2 exports z, and p_3

exports y. In addition, B has three locations ℓ_0, ℓ_1 and ℓ_2 with initial location ℓ_0. Each transition between locations has a *guard*, a *port* and an update function or the computation to be applied. For example, the transition between locations ℓ_1 and ℓ_2 is labeled by port p_2 and guarded by $x < y$ and applies computation ($x = x + y$) when executed. When this transition is executed, the value of z exported by p_2 is changed according to the valuation received through p_2.

2.2 Composite Components

We consider a set of atomic components $\{B_i\}_{i \in I}$ with $I \subseteq [1, n]$ and $B_i = \left\langle P_i, L_i, T_i, X_i, \{g_\tau\}_{\tau \in T_i}, \{f_\tau\}_{\tau \in T_i} \right\rangle$, where atomic components have disjoint sets of locations, variables and ports, i.e., for all $i, j \in I$ such that $i \neq j$, $L_i \cap L_j = \emptyset$, $P_i \cap P_j = \emptyset$ and $X_i \cap X_j = \emptyset$. We denote the set of all ports (resp. locations, variables) of a composite component by $P = \bigcup_{i \in I} P_i$ (resp. $L = \bigcup_{i \in I} L_i$, $X = \bigcup_{i \in I} X_i$). Atomic components synchronize and exchange data through interactions.

Definition 4 (Interaction). *An interaction is defined as a tuple $a = \langle P_a, G_a, F_a \rangle$, where:*

- *P_a is a non-empty set such that $P_a \subseteq P$, and, for every $i \in I$ $|P_i \cap P_a| \leq 1$, i.e., an interaction a consists of at most one port of every atomic component in B;*
- *G_a is a guard over valuation of X_a, where X_a are the variables attached to ports P_a; and*
- *F_a is an update function over the valuation of X_a.*

We denote the ports associated in an interaction a as $P_a = \{p_i\}_{i \in I}$ where i is the identification index of the atomic component because at most one port of every atomic component can be included in the same interaction. Moreover, an interaction can include variables that are denoted as $X_a = \bigcup_{p \in P_a} X_p$. The updated value of X_{p_i}, transferred to B_i as an interaction outcome, after projecting the update function F_a is denoted as F_{a_i}.

Definition 5 (Composite component). *A composite component C consists in applying a set of interactions γ to a set of distinct atomic components $\{B_i\}_{i \in I}$ with $I \subseteq [1, n]$. Therefore, a composite component C is defined as $\gamma(\{B_i\}_{i \in I})$*

Figure 2 shows an example of a composite component $C = \gamma(\{B_1, B_2, B_3\})$ where B_1, B_2 and B_3 are atomic components, and $\gamma = \{a_1, a_2, a_3, a_4, a_5\}$.

Definition 6 (Semantics of composite components). *A state q of composite component $C = \gamma(\{B_1, \ldots, B_n\})$ is an n-tuple $\langle q_1, \ldots, q_n \rangle$ where $q_i = \langle l_i, v_i, p_i \rangle$ is a state of B_i. The semantics of C is an LTS $S_c = \langle Q, \gamma, \longrightarrow \rangle$, where:*

- *$Q = B_1.Q \times \ldots \times B_n.Q$;*
- *γ is the set of all possible interactions; and*

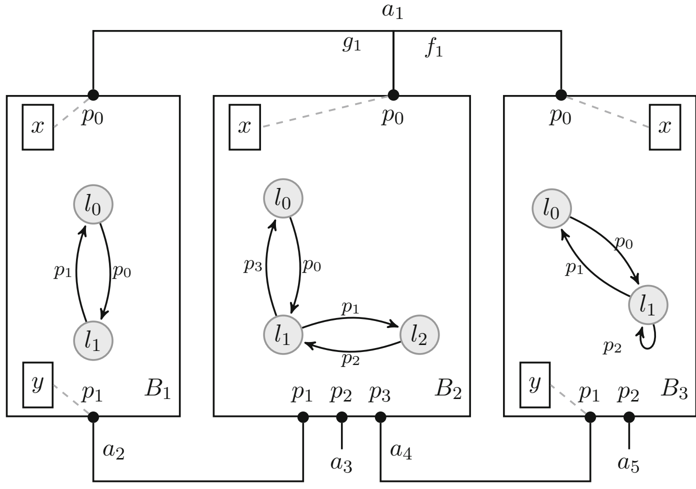


Fig. 2. Composite component in BIP

– $\longrightarrow$ *is the least set of transitions satisfying the following rule:*

$$\frac{\exists a \in \gamma : a = \left\langle \{p_i\}_{i \in I}, G_a, F_a \right\rangle \quad G_a(v(X_a)) \qquad \forall i \in I : \; q_i \xrightarrow{p_i(v_i)}_i q_i' \wedge v_i = F_{a_i}(v(X_a)) \qquad \forall i \notin I : \; q_i = q_i'}{\langle q_1, \ldots, q_n \rangle \xrightarrow{a} \langle q_1', \ldots, q_n' \rangle}$$

X_a is the set of variables attached to the ports of a, v is the global valuation. F_{a_i} is the projection of F to the variables of p_i yielding to the valuation v_{p_i} of the variables in X_i exported by p_i.

The above rule means that whenever all the ports of an interaction a are enabled and the guard corresponding to a, $(G_a(v(X_a)))$ holds, a is enabled. One enabled interaction is selected and the state of the components whose ports are involved in the interaction a changes by executing location function and moving to the next set of locations. The state of the components that are not involved in this interaction remain unchanged. A straightforward implementation of this semantics can be realized by a centralized engine that allows the execution of one enabled interaction at a time. Note, practically, it is also possible to concurrently execute independent interactions (which do not share components), while preserving the above semantics.

Figure 2 represents a composite component $\mathcal{C}$ made up of three components *atomic* $= \{B_1, B_2, B_3\}$ by applying a set of five interactions $\gamma = \{a_1, a_2, a_3, a_4, a_5\}$. For instance, interaction a_1 is enabled when all of its involved ports, *i.e.*, $B_1.p_0, B_2.p_0$ and $B3.p_0$, are enabled and its corresponding guard g_1 holds. But, in this example, the ports are not associated with guards which means that by default all ports are enabled. Assuming that guard of a_1 holds, this interaction is said to be enabled. In case it is selected to execute, its func-

tion f_1 is also applied upon its execution. Furthermore, upon the execution of a_1, transitions $\langle B_1.l_0, B_1.p_0, B_1.l_1\rangle$, $\langle B_2.l_0, B_2.p_0, B_2.l_1\rangle$, $\langle B_3.l_0, B_3.p_0, B_3.l_1\rangle$ will, also, execute for their ports are involved in a_1.

2.3 Distributed Implementation - Send/Receive BIP

A high-level BIP model can be transformed into a distributed implementation to achieve parallelism between components and interactions [6]. To do so, a BIP model is transformed into its equivalent send/receive BIP. Send/receive BIP consists of three layers: (1) an atomic components layer that consists of atomic components transformed to interact with the upper layer to execute multiparty interactions; (2) an interaction layer that consists of components responsible to execute interactions; (3) a conflict resolution layer that is responsible to forbid the concurrent execution of two conflicting interactions (to preserve the semantics of the initial model). The obtained model consists of transforming multiparty interactions into send/receive communication protocols. More precisely, each transition of atomic components is split in two transitions: (1) send offering, which sends the enabled ports to the components (that are handling the interactions corresponding to enabled ports) in the interaction layer; (2) receive, which waits for an acknowledgment from the interaction layer to execute the selected port. As such, the interaction protocol collects all enabled ports and determines what are the enabled interactions. As the interaction layer consists of several components handling different interactions, it is possible that two conflicting interactions are marked to be enabled by different components of the interaction layer, which may lead to the concurrent execution of two conflicting interactions. To remedy this, the interaction layer consults first with the conflict resolution, which is responsible for handling conflicts between interactions. Note, two interactions are said to be conflicting iff either: (1) there is a common port involved in them, or (2) if they include two distinct ports belonging to the same component where those ports are the label of two distinct transitions outgoing from the same source location.

Remark. Implementing a system with multiparty interactions requires solving potential conflicts, which is addressed in [6] for systems without priorities and in [7] for systems with priorities. Independently, we focus on those interactions that can be realized by asynchronous send/receive communication over multiparty interaction. For the sake of simplicity, and without lost of generality we consider systems without priorities.

3 Heterogeneous Distributed BIP - HDBIP

BIP uses multiparty interactions to model communication and synchronization between components, which is expressive enough to model any communication or synchronization primitives [5]. Nonetheless, modeling a simple asynchronous send/receive primitive requires to (1) explicitly create components representing buffers; (2) create intermediate schedulers to coordinate the execution of

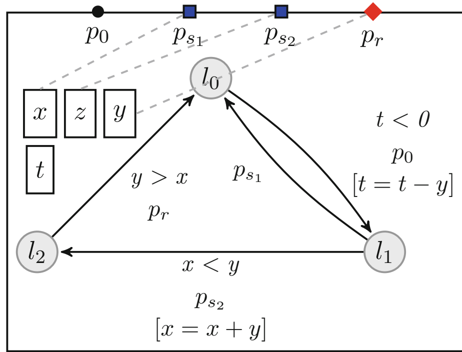


Fig. 3. Atomic component in HDBIP (Color figure online)

the interactions. This may drastically affect the performance of the generated distributed implementations. To overcome this, we introduce HDBIP that combines both multiparty and direct asynchronous send/receive (DASR) interactions. This simplifies the modeling of distributed systems and allows for efficient code generation. For instance, implementing DASR primitives can benefit from the underlying primitives such as system buffers and does not require to create extra components for scheduling with other interactions (i.e., conflict-resolution) or for buffer modeling. The components composing the HDBIP model are known as partially asynchronous (PA) atomic components.

3.1 HDBIP Syntax

A PA atomic component $B^\star$ is a regular BIP atomic component where transitions are labeled with three types of ports: ordinary, direct send and direct receive: (1) ordinary ports are the same to those defined in BIP; (2) direct send ports are used to model asynchronous direct communication with receive ports. Hereafter, we represent ordinary, direct send and direct receive ports, by black circle, blue rectangle and red diamond, respectively.

Definition 7 (Partially Asynchronous Atomic component). *A PA atomic component $B^\star$ is tuple $\langle B, t\rangle$ where:*

- *B is an atomic component;*
- *$t : P \to \{ordinary, send, receive\}$ is a function that maps ports to their types.*

Figure 3 depicts a PA atomic component $B^\star$ in HDBIP. $B^\star$ has four ports p_0, p_{s_1}, p_{s_2}, and p_r and four local variables x, y, z, and t. Port p_{s_1} exports x, p_{s_2} exports z and p_r exports y. In addition, B has three locations ℓ_0, ℓ_1 and ℓ_2 with initial location ℓ_0. Hereafter, we consider a set of PA atomic components $\{B_i^\star\}_{i\in I}$, where $\forall i \in I$, $B_i^\star = \langle B_i, t_i\rangle$. Let $P_o = \bigcup_{i\in I} B_i^\star.P_o$ (resp. P_s, P_r, P) denotes the set of all the ordinary (resp. direct send, direct receive, all) ports.

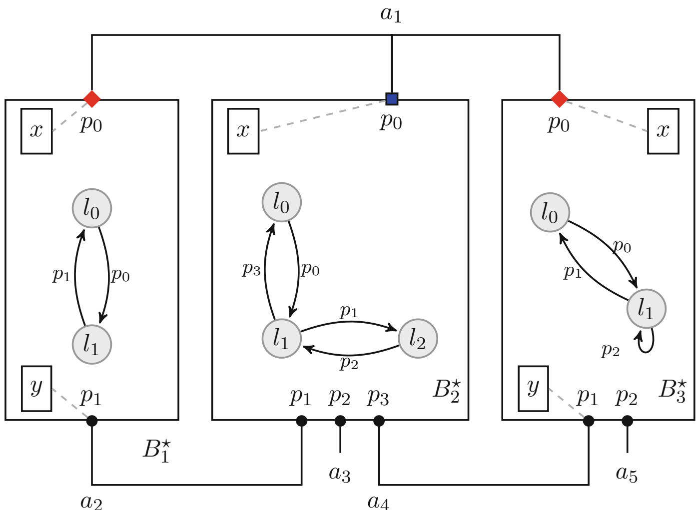


Fig. 4. Composite component in HDBIP

Moreover, without loss of generality, we assume that from any location, the outgoing transitions can be labeled with both ordinary and send or receive ports (i.e., either only ordinary ports or a mix of send or receive ports). This allows to efficiently generate distributed implementation and makes the interaction model not ambiguous to the developers of the atomic components. Note that the transitions requirements hold in the PA component depicted in Fig. 3.

We distinguish two types of interactions: (1) ordinary; and (2) DASR. Ordinary interaction is the same as regular BIP interaction, i.e., allows to model multiparty interaction. Hence, it connects ordinary ports. DASR interaction allows to model asynchronous send receive interaction and connects a sender port of a component to receiver ports of different components.

Definition 8 (Ordinary Interaction). *An ordinary interaction a is defined by the tuple* $\langle P_a, G_a, F_a \rangle$ *where:*

- $P_a \subseteq P$ *is a non-empty set such that* $P_a \subseteq P_o$ *and,* $\forall i \in I$, $|B_i^\star.P \cap P_a| \leq 1$*; and*
- G_a *and* F_a *are the guard and the function of the ordinary interaction, the same as the ones defined in the BIP interaction.*

Definition 9 (DASR Interaction). *A DASR interaction a is defined by* P_a *where:*

- $P_a \subseteq P$*, with* $|P_a| > 1$*, is a set such that* $|P_a \cap P_s| = 1$*,* $|P_a \cap P_o| = 0$*,* $|P_a \cap P_r| > 0$ *and,* $\forall i \in I, |P_i \cap P_a| \leq 1$*;*
- *all ports have the same type; (3) its guard is always hold, however, its send port can have a local guard;*

– *its function allows only for data transfer of data attached to the sender port to the data attached to the receiver ports.*

Note that a send port can only participate in one DASR interaction, whereas a receive port can participate in several DASR interactions.

Definition 10 (Partially asynchronous composite component). *A PA composite component $C^\star$ denoted by $\gamma^\star(\{B_i^\star\}_{i\in I})$ consists of a set of atomic components $\{B_i^\star\}_{i\in I}$ composed by applying a set of ordinary and DASR interactions $\gamma^\star$.*

Given a PA composite component $\gamma^\star(\{B_i^\star\}_{i\in I})$ where $B_i^\star = \langle B_i, t_i\rangle$ for all $i \in I$, we define:

- $type : \gamma^\star \rightarrow \{ordinary, sendreceive\}$ is a function that maps interactions to their types;
- $\gamma_o = \{a \in \gamma^\star \mid type(a) = ordinary\}$ the set of all ordinary interactions; and
- $\gamma_{sr} = \{a \in \gamma^\star \mid type(a) = sendreceive\}$ the set of all DASR interactions.

Clearly, $\gamma^\star = \gamma_o \cup \gamma_{sr}$ and $\gamma_o \cap \gamma_{sr} = \emptyset$. Figure 4 depicts a PA composite component made up of a set of three PA components $B^\star = \{B_1^\star, B_2^\star, B_3^\star\}$ by applying a set of five interactions $\gamma^\star = \{a_1, a_2, a_3, a_4, a_5\}$, where only a_1 is a DASR interaction while the rest (a_2, a_3, a_4 and a_5) are ordinary interactions. a_1 is a DASR interaction because it consists of a direct send port $B_2^\star.p_0$ and the receive ports $B_1^\star.p_0$ and $B_3^\star.p_0$. a_1 is said to be a valid DASR interaction because it does not include any ordinary port. Moreover, $B_2^\star.p_0$ cannot participate in further interactions.

3.2 HDBIP Semantics

We define the semantics of a PA composition component $C^\star$ by transforming it into its equivalent BIP model $C = [\![C^\star]\!]$. The transformation consists of the following steps: (1) create buffer atomic components; (2) create interactions connecting send and receive ports with buffer components.

Creating Buffer Components. We first create a buffer component Bu_p^i for every receive port $p \in \bigcup_{i\in I} B_i^\star.Pr$. Bu^i is an atomic component where: (1) $Bu.X = X_s \cup X_r \cup D$ such that D is a queue that can hold data of the same type of the port, $X_s = \{x_s \mid x \in p.X\}$ is the set of received variables that correspond to port p, and $X_r = \{x_r \mid x \in p.X\}$ is the set of send variables that correspond to port p; (2) $Bu.P = \{send, receive\}$, where *send* port exports the set of variables X_s, and *receive* port exports the set X_r; (3) $Bu.L = \{l_0, l_1\}$, where l_0 is the initial location; (4) $Bu.T = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ such that $\tau_1 = \langle l_0, receive, l_1\rangle$, $\tau_2 = \langle l_1, receive, l_1\rangle$, $\tau_3 = \langle l_1, send, l_1\rangle$ and $\tau_4 = \langle l_1, send, l_0\rangle$. The guards of transitions are predicates over the queue D and its size. Assuming the queue size can be denoted as $D.size$, guard g_1 of τ_1 is $D.size = 0$, g_2 of τ_2 is $D.size > 0$, g_3 of τ_3 is $D.size > 1$, and, finally, g_4 of τ_4 is $D.size = 1$. Yet, the size of the queue

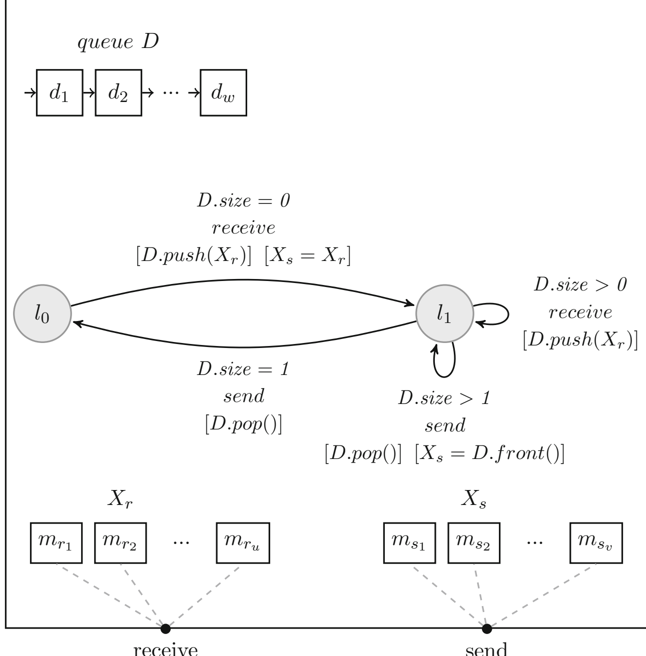


Fig. 5. Buffer component

is not determined by the size of the message received, but by the number of messages received. The functions, on transitions including the port receive, from l_0 to l_1, involve adding the values of X_r (as one list) to the list D and updating the values of X_s to that of X_r, whereas, from l_1 to l_1, the values of X_r are only added to D. Initially, in τ_1, X_s is updated to the values of the first received message. Thus, the functions, on transitions including the port send, from l_1 to l_1, involve removing data from the list D first, then updating the values of X_s to be the oldest list of values received and pushed to D. On the other hand, from l_1 to l_0, only the last list of values in D is removed emptying D. The set of all buffers for all receive ports in $C^\star$ is denoted by $BU = \bigcup_{i \in I} \{Bu_p^i \mid p \in B_i^\star.P \wedge t_i(p) = \mathit{receive}\}$. Note that port *receive* of the buffer is always enabled, i.e., its guard is `true`, whereas port *send* is enabled when there are messages to be sent, i.e., the internal queue is not empty. Figure 5 shows an example of a buffer component that corresponds to port $p[X_r]$ (port p exporting a set of variables X_r).

Integration. We now are ready to define the semantics of a partially asynchronous composite component $C^\star$ as follows: (1) create a buffer component for each receive port; (2) append ordinary interactions; (3) for each DASR interaction we create one interaction connecting the send port of the DASR interaction

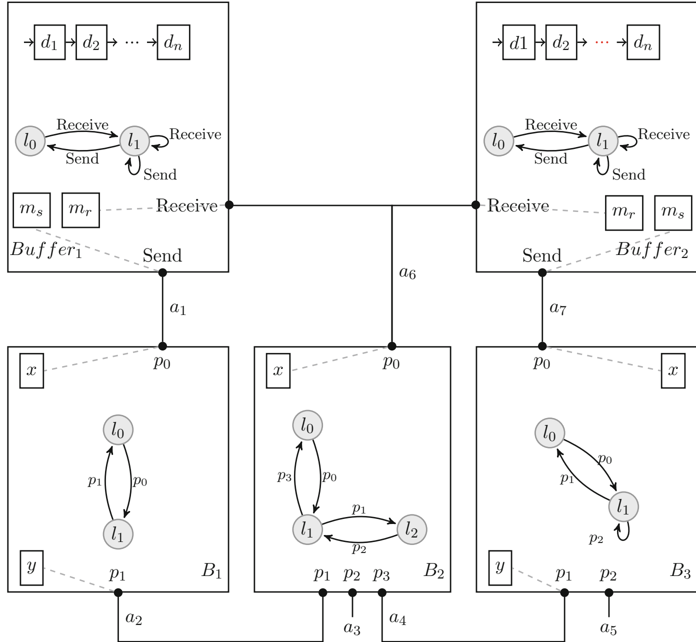


Fig. 6. Transformation of HDBIP composite in Fig. 4 to BIP

to the receive ports of the buffers that correspond to the receive ports of the DASR interaction, and we create one binary interaction for each receive port, which is connected to send port of its corresponding buffer. Finally, all send and receive ports in HDBIP become ordinary.

Definition 11 (PA composite component semantics). *Given a partially asynchronous composite component $C^\star = \gamma^\star(\{B_i^\star\}_{i\in I})$, its semantics is defined by the transformation into a regular BIP system $C = [\![C^\star]\!]$, such that $C = \gamma(\{B_i\}_{i\in I} \cup BU)$ where:*

- *B_i is the atomic component that corresponds to $B_i^\star$ by removing labeling of the ports;*
- *BU is the set of buffers created for each receive port in $C^\star$;*
- *γ is the set of interactions applied to the set of atomic components $\{B_i\}_{i\in I} \cup BU$ such that $\gamma = \gamma_o \cup \gamma_s \cup \gamma_r$ where, γ_o is the set of all ordinary interactions in $C^\star$, $\gamma_s = \bigcup_{a\in\gamma_{sr}}\{(P_a, \texttt{true}, \texttt{identity}) \mid P_a = \{a.send\} \cup \bigcup_{r\in a.recvs}\{Bu_p^i.recv\}\}$ is the set of interactions between each direct send port in interaction a and the corresponding buffer receive port, and $\gamma_r = \bigcup_{p\in P_r}\{(P_a, \texttt{true}, \texttt{identity}) \mid p \in B_i^\star.P \wedge P_a = \{p, Bu_p^i.send\}\}$ is the set of interactions between each receive port and its corresponding send buffer port.*

Figure 6 shows how the HDBIP model presented in Fig. 4 is transformed to its equivalent BIP model. All the PA atomic components $B_1^\star, B_2^\star$ and $B_3^\star$ are transformed to their equivalent BIP versions (ignoring ports types) B_1, B_2 and B_3

respectively. For every direct receive port ($B_1^\star.p_0$, $B_3^\star.p_0$) we added a corresponding buffer component in the BIP model. Then, the DASR interaction, in the HDBIP model, from the send port $B_2^\star.p_0$ to $B_1^\star.p_0$ and $B_3^\star.p_0$ is replaced by an interaction involving $B_2.p_0$ and the port receive of each of the buffer components corresponding to the receive ports of the HDBIP model. Additionally, DASR replacement includes adding other interactions involving the port send of every buffer component and its corresponding previous receive port. For this example, we included two interactions: (1) involving $B_1.p_0$ and *Buffer1.send*, (2) $B_3.p_0$ and *Buffer2.send*.

4 Efficient Code Generation

Given an HDBIP system, it is possible to transform it to a regular BIP (i.e., consisting only of regular ports) and use the code generation provided by BIP (three-layer model). However, this may lead to the generation of inefficient implementations mainly because of: (1) the buffer components that correspond to receive ports will be replaced with actual threads or processes; (2) interactions between send/receive ports and the buffer components will be mixed with the multiparty interactions and will be added to the interaction protocol components; hence, their execution requires communication between base components, interaction protocols and possibly with conflict resolution components in case of conflicts. Although using HDBIP simplifies the development process by automatically generating buffer components and the corresponding communications, a naive implementation would impose an additional overhead due to the extra communication as well as the creation of unnecessary threads/processes to represent the buffer components. Therefore, we introduce an efficient code generation that allows to avoid the creation of buffer components and the communication with the interaction and conflict resolution layers. To do so, we first transform PA atomic components of HDBIP system by splitting (following [6]) the transitions labeled with ordinary ports into two transitions to interact and receive notifications from the interaction protocol components, respectively. As for the transitions labeled with send and receive ports are not split and kept unchanged. Figure 7 presents an example of the transformation of a PA atomic component into its equivalent PA send/receive atomic component. Second, we generate the three layer send/receive model by only creating components in the interaction layer for ordinary interactions. DASR interactions are not integrated with the interaction layer and remain in the transformed model.

4.1 Correctness

The aim of the proof is to show that the efficient code generation is equivalent to the one provided by transforming HDBIP into regular BIP. The proof consists of two independent steps: (1) preservation of the buffer components; (2) no need for conflicts handling.

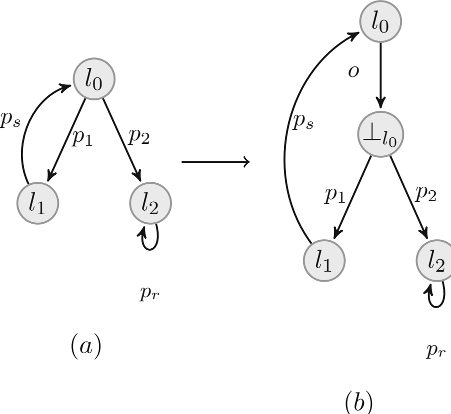


Fig. 7. HDBIP transitions transformation in 3-layer model

Preservation of the Buffer Components. Our code generation produces C++ implementation that uses MPI for the communication between threads/processes. As MPI has its internal system buffer, sending a message to a specific receive port (i.e., labeled with the name of the receive port) is implicitly added to the system buffer of MPI with the corresponding label. As such, there is no need to create buffer components.

No Need for Conflict Handling. In the equivalent BIP model obtained from HDBIP (Definition 11), conflicts may occur between: (1) only ordinary interactions; (2) ordinary and DASR interactions; (3) only direct/send interactions. Recall that our efficient code generation only requires to integrate ordinary interactions into interaction and conflict resolution layer, whereas DASR interactions are kept unchanged in the 3-layer send/receive model. As such, conflicts between only ordinary interactions are resolved by the interaction protocol and the conflict resolution protocol layers in the usual way (Sect. 2). As from any state, the outgoing transitions can be labeled with either ordinary ports or send/receive ports, it is not possible to get conflicts between ordinary and direct send/receive interactions. Regarding direct send/receive interactions, a conflict may arise between two interactions that either involve: (1) a common direct receive port; (2) a common direct send port; (3) two ports of the same component that are the labels of two outgoing transitions from a same state. As for the (1) the execution of the receive port allows the buffer component to remain in state l_1 (see Fig. 5). As such, even in the case of two concurrently-executing interactions connected to the same receive port, the final state will still belong to the state space of the semantics of the transformed regular BIP. As for (2) a direct send port can be connected to only one interaction. As for (3) we consider several cases either: (3a) the two ports are send ports, then the component will pick one of the two ports and execute the corresponding send; (3b) the two ports

are receive ports, then the component can execute the corresponding receive port that has a message on its buffer, that is; (3c) one port is send and another is receive, in order to avoid deadlock of the execution, we consider giving priority to send port if its guard is enabled, otherwise, we can safely wait until a message on one of the receive ports is available. Consequently, in all the cases a conflict can be resolved locally.

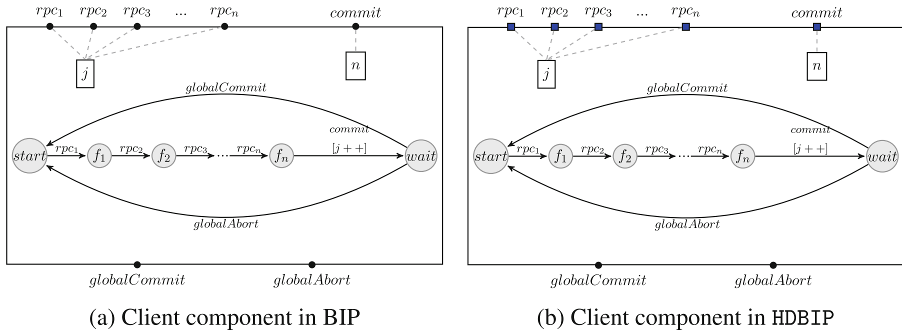
5 Performance Evaluation

We evaluate the execution times and the number of lines of code in `HDBIP` versus BIP on distributed two-phase commit protocol [13]. Two-phase commit is a consensus protocol used to commit or abort a distributed transaction. A distributed transaction consists of a sequence of operations applied to several processes/participants. The system consists of n resource managers (participant of the transaction) $rm_1, rm_2, \ldots, rm_n$ and a transaction manager tm. Executing a distributed transaction consists of the following steps: (1) the client sends a begin transaction message to tm; (2) client executes the operations of the transaction on its participants (resource managers); (3) client sends a commit transaction message to tm; (4) tm starts running two-phase commit protocol by sending a vote request message to all the resource managers; (5) each resource manager has the ability to commit or abort the transaction by sending local commit or local abort; (6) tm receives all the votes and broadcasts global commit to all resource managers if it has received a local commit from all the resource managers, otherwise it broadcasts global abort message; finally (7) depending on the receive message a resource manager either aborts or commits the transaction. For the sake of simplicity, we omit the handling of crash/recovery and timeouts that are handled by running specific termination protocols and by assuming the existence of persistent storage to keep track of the logs.

We provide two implementations of two-phase commit protocol using standard BIP and `HDBIP`. Figures 8a and b show the atomic components of the clients in standard BIP and `HDBIP`, respectively. It mainly initiates the transaction by calling remote procedure calls on the resource managers accompanied with the current transaction id j. It then notifies tm through the port *commit* and waits for the reception of the global decision. In case of standard BIP all ports are ordinary. In `HDBIP` only *globalAbort* and *globalCommmit* ports are ordinary as they require a global agreement (multiparty interaction), and all the other remaining ports are send ports.

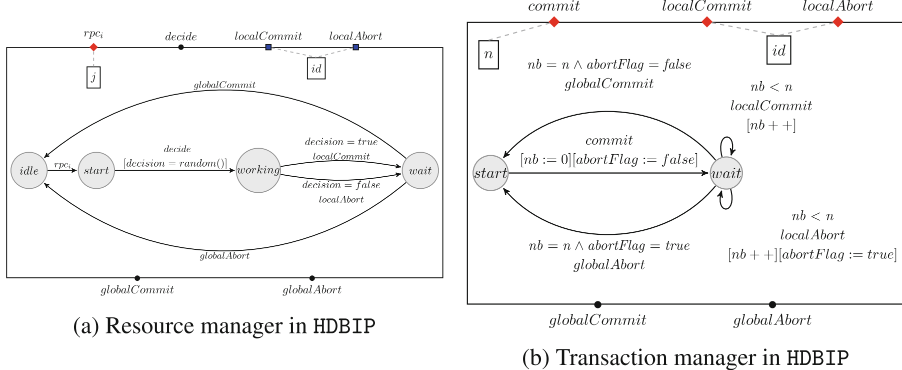
The behavior of the resource manager rm and transaction manager tm in `HDBIP` are shown in Figs. 9a and b (in regular BIP, we have the same behavior but all ports are ordinary). Each rm starts the transaction by executing the function. Then, a decision is made to abort or commit the transaction. Accordingly, it either synchronizes with tm with the port *localCommit* or *localAbort*. tm collects all the responses and synchronizes with all the resource managers as well as the clients to globally commit or abort the transaction.

Figures 10 and 11 show the composite component of the whole system in regular BIP and `HDBIP`, respectively. Recall that in regular BIP all buffers should



(a) Client component in BIP (b) Client component in HDBIP

Fig. 8. Client component in BIP and HDBIP



(a) Resource manager in HDBIP

(b) Transaction manager in HDBIP

Fig. 9. Resource and transaction managers in HDBIP

be explicitly modeled with components and all ports are ordinary ports. In HDBIP the design is much simpler as buffer components will be implicitly replaced by the system buffers during code generation.

Efficiency. We compare the execution times of the distributed implementations generated from BIP and the one generated from HDBIP. Note that in case of HDBIP the direct send receive interactions are treated in a special way and are not integrated with the regular code generation of multiparty interactions. We consider two different scenarios by varying the number of resource managers and the number of transactions. For both scenarios, we consider a cluster of four Linux machines (64-bit Ubuntu 16.04), each with 8 cores, Intel Core i7-6700 processor, and 32 GB memory. In the first scenario, we vary the number of transactions from 20,000 to 200,000 by a step of 20,000 and we fix the number of resource managers to be 10. In the second scenario, we vary the number of resource managers from 2 to 20 by a step of 2, and we fix the number of transactions to be 10,000. Figures 12a and b show the execution times of theses scenarios for both implementations, respectively. In both scenarios, it is clear

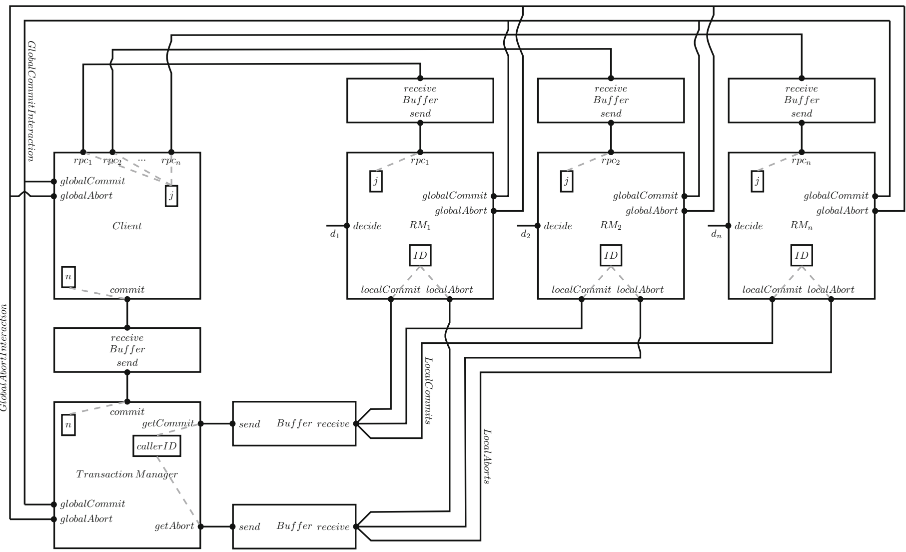


Fig. 10. Two-phase commit in BIP

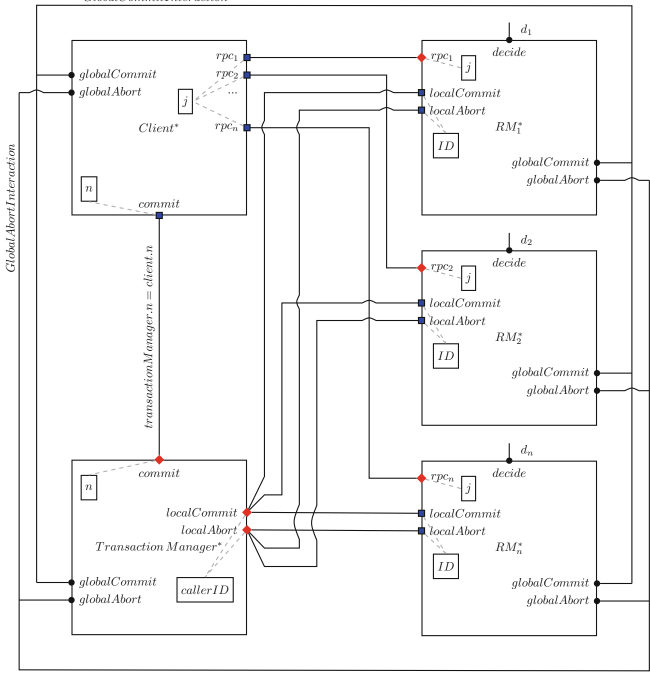
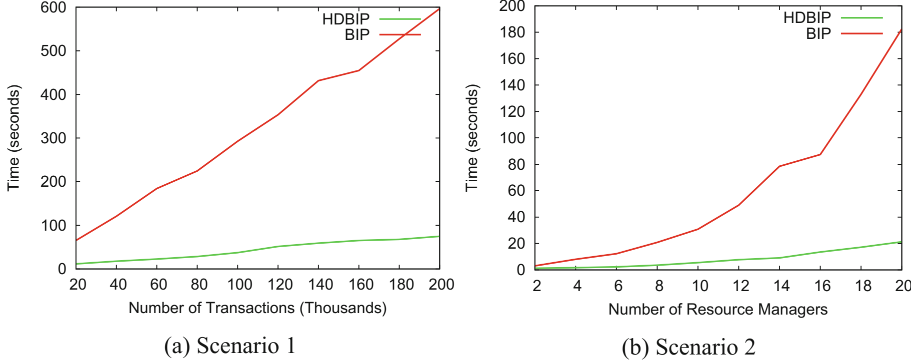


Fig. 11. Two-phase commit in HDBIP

that the implementation of HDBIP drastically outperforms regular BIP. This is mainly due to the extra messages exchanged in case of the regular BIP with the buffer components, and the multiparty interactions between the buffer components. In case of HDBIP, we can still execute multiparty interactions, however, direct send receive can be directly executed with no need to create message buffer and benefit from the system buffers that are already available.



(a) Scenario 1

(b) Scenario 2

Fig. 12. Performance evaluation of two-phase commit

Lines of code (LOC). Using HDBIP requires less LOC than BIP as there is no need to create (1) the buffer component type and the corresponding instances; (2) the interactions between send/receive ports and the buffer components. For instance, modeling two-phase commit in case of 10 resource managers, requires 280 LOC in case of HDBIP and 390 LOC in case of BIP.

6 Related Work

In [11], a method is introduced to automatically generate correct asynchronously communicating processes starting from a global communication protocol. Unlike our model, the proposed method considers a simple communication model where each message has a unique sender and receiver. As such, modeling multiparty interactions requires to explicitly defining the communication protocol and conflict resolution handling, which is time consuming and error prone.

Session types [4,8,12,15,17] model interactions between distributed processes, and are based on the following methodology: (1) interactions are described as a *global protocol* between processes; (2) *Local protocols* are synthesized by projecting global protocol to local processes; (3) implementation of local processes; (4) type-checking of local types with respect to local processes. The design methodology of session type has major drawbacks: (1) there is a huge

gap between design and implementation; (2) the design flow includes redundancy (global protocol, local protocol, process implementation), which is error prone; (3) there is no clear separation between communication and computation in local processes.

LASP [16] is a programming model designed to facilitate the development of reliable and large-scale distributed computing. It combines ideas from deterministic data-flow programming and conflict-free replicated data types (CRDTs). However, LASP is tailored to consistency over replicated data types. It would be interesting to integrate LASP with HDBIP to support fault-tolerance in HDBIP.

Other industrial frameworks simplify the development of large scale distributed systems such as AzureBot [1]. However, using such frameworks modeling communication models and synchronization are too abstract, which does not allow the expressiveness of explicit communication models. Moreover, AzureBot supports only applications written in C# and hosted in the Azure cloud platform.

Some recent research efforts tackle correctness-preserving code generation from models to asynchronously communicating systems. For example, in AlbertBBM16,HenrioR16, they introduce a formal translation from abstract behavioral specification (ABS) to object-oriented implementation, where [2] (resp. [14]) specifically targets parallel (resp. distributed) systems. However, the underlying communication model of ABS does not support multiparty interactions but only asynchronous calls.

7 Conclusion and Perspectives

We introduce a rigorous model to facilitate the development of correct, efficient and scalable distributed systems. In particular, HDBIP allows both multiparty and asynchronous send/receive primitives. Moreover, our method (1) uses the primitives provided by the underlying systems such as system buffers; and (2) makes a clear separation, which is correct-by-construction, between multiparty interactions and asynchronous send/receive interactions; which allow the generation of efficient distributed implementations

For future work, we first consider to develop a source-to-source transformation from session types to HDBIP. This would avoid code redundancy of the methodology provided by session types. Moreover, we consider using other primitives provided by the underlying library (e.g., MPI) such as barriers in order to support efficient implementation of multiparty interactions. We also work on extending HDBIP to support fault tolerance. We also consider to leverage the asynchronous send/receive communication primitive to improve the efficiency of the runtime verification [10] and enforcement [9] of component-based systems.

Acknowledgment. The authors acknowledge the support of the University Research Board (URB) at American University of Beirut and the ICT COST (European Cooperation in Science and Technology) Action IC1402 Runtime Verification beyond Monitoring (ARVI).

References

1. Agarwal, D., Prasad, S.K.: AzureBOT: a framework for bag-of-tasks applications on the azure cloud platform. In: 2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Ph.D. Forum (2013). https://doi.org/10.1109/ipdpsw.2013.261
2. Albert, E., Bezirgiannis, N., de Boer, F., Martin-Martin, E.: A formal, resource consumption-preserving translation of actors to Haskell. In: Hermenegildo, M., Lopez-Garcia, P. (eds.) LOPSTR 2016. LNCS, vol. 10184, pp. 21–37. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63139-4_2
3. Basu, A., et al.: Rigorous component-based system design using the BIP framework. IEEE Softw. **28**(3), 41–48 (2011)
4. Bejleri, A., Yoshida, N.: Synchronous multiparty session types. Electron. Notes Theor. Comput. Sci. **241**, 3–33 (2009). https://doi.org/10.1016/j.entcs.2009.06.002
5. Bliudze, S., Sifakis, J.: The algebra of connectors - structuring interaction in BIP. IEEE Trans. Comput. **57**(10), 1315–1330 (2008). https://doi.org/10.1109/TC.2008.26
6. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: A framework for automated distributed implementation of component-based models. Distrib. Comput. **25**(5), 383–409 (2012). https://doi.org/10.1007/s00446-012-0168-6
7. Bonakdarpour, B., Bozga, M., Quilbeuf, J.: Model-based implementation of distributed systems with priorities. Des. Autom. Embed. Syst. **17**(2), 251–276 (2013). https://doi.org/10.1007/s10617-012-9091-0
8. Bonelli, E., Compagnoni, A.: Multipoint session types for a distributed calculus. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 240–256. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78663-4_17
9. Falcone, Y., Jaber, M.: Fully automated runtime enforcement of component-based systems with formal and sound recovery. STTT **19**(3), 341–365 (2017). https://doi.org/10.1007/s10009-016-0413-6
10. Falcone, Y., Jaber, M., Nguyen, T., Bozga, M., Bensalem, S.: Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. Softw. Syst. Model. **14**(1), 173–199 (2015). https://doi.org/10.1007/s10270-013-0323-y
11. Farah, Z., Ait-Ameur, Y., Ouederni, M., Tari, K.: A correct-by-construction model for asynchronously communicating systems. Int. J. Softw. Tools Technol. Transf. **19**(4), 465–485 (2017)
12. Gay, S.J., Vasconcelos, V.T., Ravara, A., Gesbert, N., Caldeira, A.Z.: Modular session types for distributed object-oriented programming. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, 17–23 January 2010, pp. 299–312 (2010). https://doi.org/10.1145/1706299.1706335
13. Gray, J., Lamport, L.: Consensus on transaction commit. ACM Trans. Database Syst. **31**(1), 133–160 (2006). https://doi.org/10.1145/1132863.1132867
14. Henrio, L., Rochas, J.: From modelling to systematic deployment of distributed active objects. In: Lluch Lafuente, A., Proença, J. (eds.) COORDINATION 2016. LNCS, vol. 9686, pp. 208–226. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39519-7_13

15. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, USA, 7–12 January 2008, pp. 273–284 (2008). https://doi.org/10.1145/1328438.1328472
16. Meiklejohn, C., Van Roy, P.: Lasp: a language for distributed, coordination-free programming. In: Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, PPDP 2015, pp. 184–195. ACM, New York (2015). https://doi.org/10.1145/2790449.2790525
17. Vallecillo, A., Vasconcelos, V.T., Ravara, A.: Typing the behavior of software components using session types. Fundam. Inform. **73**(4), 583–598 (2006). http://iospress.metapress.com/content/82bf1qafeel5g8n4/

State-of-the-Art Model Checking for B and Event-B Using ProB and LTSmin

Philipp Körner[1(✉)], Michael Leuschel[1], and Jeroen Meijer[2]

[1] Institut für Informatik, Universität Düsseldorf, Düsseldorf, Germany
p.koerner@uni-duesseldorf.de, leuschel@cs.uni-duesseldorf.de
[2] Formal Methods and Tools, University of Twente, Enschede, The Netherlands
j.i.g.meijer@utwente.nl

Abstract. In previous work, we presented symbolic reachability analysis by linking ProB, an animator and model checker for B and Event-B, and LTSmin, a language-independent model checker offering state-of-the-art model checking algorithms. Although the results seemed very promising, it was a very basic integration of these tools and much potential of LTSmin was not covered by the implementation.

In this paper, we present a much more mature version of this tool integration. In particular, explicit-state model checking, efficient verification of state invariants, model checking of LTL properties, as well as partial order reduction and proper multi-core model checking are now available. The (improved) performance of this advanced tool link is benchmarked on a series of models with various sizes and compared to ProB.

1 Introduction

Formal methods, e.g., the B-Method [3], are vital to ensure correctness in software where failure means loss of money or even risking human lives. Yet, for industrial application, tooling often remains unsatisfactory [6,34]. One such tool is ProB, an animator and model checker for B and Event-B. While ProB is fairly mature after hundreds of man-years of engineering effort, it may still struggle with industrial-sized models containing several millions of states. LTSmin, however, is a language-independent model checker that offers symbolic algorithms and many optimizations in order to deal with the state space explosion problem.

In [4], we linked LTSmin with ProB in order to obtain a symbolic reachability analysis for B and Event-B. ProB was computing the B operational semantics and providing static information about possible state transitions while LTSmin was performing the symbolic reachability algorithm.

LTSmin offers further model checking algorithms and optimizations, both for its symbolic and for its sequential backend. In this paper, we describe how we extended the link between LTSmin and ProB using ZeroMQ [18] in order to obtain a model checking tool for B and Event-B and evaluate the performance on a set of real life, industrial-sized models. LTSmin's language frontend that

J. Meijer—Supported by STW SUMBAT grant: 13859.

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 275–295, 2018.
https://doi.org/10.1007/978-3-319-98938-9_16

interacts with ProB can directly be used with any model checker that speaks the same protocol via ZeroMQ. The extension is based on [20] and includes:

- invariant checking (for both the symbolic and sequential backend),
- guard splitting for symbolic analysis of B models,[1]
- partial order reduction (with the sequential backend),
- (parallel) LTL model checking (with the sequential/multi-core backend),
- effective caching of transitions and state labels,
- short states, to transmit only relevant variables for each transition.

1.1 LTSmin

LTSmin [8] is an open-source, language-independent, state-of-the-art model checker that offers many model checking algorithms including partial order reduction, LTL verification and distributed model checking. An overview of its architecture can be found in Fig. 1. By implementing the Pins, i.e., the partitioned interface to the next-state function, new language frontends can employ these algorithms. At its core, Pins consists of three functions: one that provides an initial state vector, a second, partitioned transition function that calculates successor states and, lastly, a state labelling function.

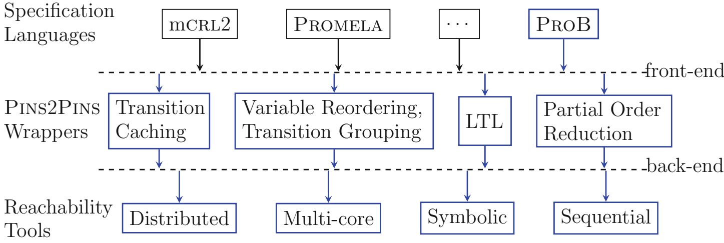


Fig. 1. Modular Pins architecture of LTSmin [19]

LTSmin provides four backends:

- a sequential backend that implements an explicit state model checking algorithm similar to the one implemented in ProB,
- a symbolic backend that stores states as LDDs (List Decision Diagrams) [7],
- a multi-core backend that works similar to the sequential backend, but is capable of using multiple CPU cores on the same machine,
- a distributed backend in order to utilize multiple machines for model checking.

For this article, we will focus on the advances of the integration with ProB using the sequential and symbolic backends but also experiment with the multi-core backend. We have not done any experiment with the distributed backend yet.

[1] Due to technical limitations in ProB, we have not added this for Event-B yet.

1.2 ProB and the B-Method

ProB [25] is an open-source animator and model checker for several formalisms including B, Event-B, CSP, Z and TLA$^+$. It can be used in order to find invariant or assertion violations or deadlock states in machine specifications. While it implements a straightforward explicit state model checker, it also ships more advanced techniques, e.g., symmetry reduction [29], partial order reduction [15,16] or symbolic model checking [22]. This style of symbolic model checking [9], where states are stored as predicates, must not be confused with the symbolic model checking that LTSmin provides, where states are stored as decision diagrams. ProB's core is written in SICStus Prolog [10] and may also employ SMT solvers [23], such as Z3 and CVC4, or SAT solvers, such as Kodkod [28].

When integrating ProB into LTSmin, we focus on two formalisms: B (sometimes referred to as "classical B") is part of the B-Method [3], where software is developed starting with a very abstract model that iteratively is refined to a concrete implementation. This method aims for software to be "correct by construction". Event-B [1] is considered to be the successor of B that does not include constructs that often hinder formal proof in the language, e.g., conditional assignments or loops. Both formalisms offer a very high level of abstraction and are based on set theory and first-order logic.

1.3 Theoretical Background

We repeat the most important definitions used in [4] on the following contrived example:

```
MACHINE example
CONSTANTS c
PROPERTIES c = 100
VARIABLES x,y
INVARIANT x : INTEGER & y : INTEGER &
          x <= c & x + y <= 2 * c
INITIALISATION x := 0 || y := 0
OPERATIONS
  incx        = SELECT x < c THEN x := x + 1 END;
  doublex     = SELECT x < c/2 & x > 0 THEN x := x * 2 END;
  incy(n)     = SELECT n > 0 & n < c & y < c
                THEN y := y + n END;
  incxmaybey  = SELECT x < c
                THEN x := x + 1 ||
                     IF x mod 2 = 0 THEN y := y + 1 END
END
```

Fig. 2. Contrived B machine example

Definition 1 (Transition System). *A Transition System (TS) is a structure* $(\mathcal{S}, \rightarrow, I)$*, where* $\mathcal{S}$ *is a set of states,* $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ *is a transition relation and* $I \subseteq \mathcal{S}$ *is a set of initial states. Furthermore, let* $\rightarrow^*$ *be the reflexive and transitive closure of* $\rightarrow$*, then the set of reachable states is* $\mathcal{R} = \{s \in \mathcal{S} \mid \exists s' \in I \,.\, s' \rightarrow^* s\}$*.*

Such transition systems are induced by both B and Event-B models. As all variables have to be typed, the set of states $\mathcal{S}$ is the Cartesian product of all types. All possible initial states are given in the `INITIALISATION` machine clause. The union of all operations define the transition relation $\rightarrow$. For symbolic model checking however, it is very important that the transition relation is split into groups.

Definition 2 (Partitioned Transition System). *A Partitioned Transition System (PTS) is a structure* $\mathcal{P} = (S^{\text{N}}, \mathcal{T}, \rightarrow^{\text{M}}, I^{\text{N}})$*, where*

- $S^{\text{N}} = S_1 \times \ldots \times S_{\text{N}}$ *is the set of states, which are vectors of* N *values,*
- $\mathcal{T} = (\rightarrow_1, \ldots, \rightarrow_{\text{M}})$ *is a vector of* M *relations, called transition groups,* $\rightarrow_i \subseteq S^{\text{N}} \times S^{\text{N}}$ $(\forall 1 \leq i \leq \text{M})$*,*
- $\rightarrow^{\text{M}} = \bigcup_{i=1}^{\text{M}} \rightarrow_i$ *is the overall transition relation induced by* $\mathcal{T}$*, i.e., the union of the* M *transition groups, and*
- $I^{\text{N}} \subseteq S^{\text{N}}$ *is the set of initial states.*

We write $\boldsymbol{s} \rightarrow_i \boldsymbol{t}$ *when* $(\boldsymbol{s}, \boldsymbol{t}) \in \rightarrow_i$ *for* $1 \leq i \leq \text{M}$*, and* $\boldsymbol{s} \rightarrow^{\text{M}} \boldsymbol{t}$ *when* $(\boldsymbol{s}, \boldsymbol{t}) \in \rightarrow^{\text{M}}$*.*

Strictly speaking, the transition relation in Definition 2 is not partitioned, as individual B operations can have same effect. We implemented an easy mental model where each transition group represents exactly one operation in the B model.

For the example in Fig. 2, the only initial state is $init^{(c,x,y)} = (100, 0, 0)$. We agree on a notation where the ordering of the variables in an individual state is given once as a superscript. In LTSmin, the ordering of the variables is fixed and unambiguous. Then, $I^{\text{N}} = \{(100, 0, 0)\}$, $S^{\text{N}} = \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ and $\mathcal{T} = (\texttt{incx}, \texttt{doublex}, \texttt{incy}, \texttt{incxmaybey})$ with, e.g., $\texttt{incx} = \{(100, 0, 0) \rightarrow_1 (100, 1, 0)\,, (100, 0, 1) \rightarrow_1 (100, 1, 1)\,, (100, 1, 0) \rightarrow_1 (100, 2, 0)\,, \ldots\}$.

Symbolic Model Checking and Event Locality. In many B models, operations only read from and write to a small subset of variables, which is known as event locality [11]. Symbolic model checking benefits from event locality, allowing reuse of successor states when only variables changed that are irrelevant to the state transition.

In order to employ LTSmin's symbolic algorithms [7,26,27], ProB provides several dependency matrices about the B model that shall be checked: A *read* matrix and *may-write* matrix is used in order to determine independence between transition groups for symbolic model checking. These two matrices for Fig. 2 are given in Fig. 3. Entries are set to 1, if the operation reads or writes the variable, and otherwise to 0. Further matrices are shown once their use-case is introduced.

$$\begin{array}{l} \\ \texttt{incx} \\ \texttt{doublex} \\ \texttt{incy} \\ \texttt{incxmaybey} \end{array} \begin{array}{c} \begin{array}{ccc} \texttt{c} & \texttt{x} & \texttt{y} \end{array} \\ \begin{bmatrix} \mathbf{1} & \mathbf{1} & 0 \\ \mathbf{1} & \mathbf{1} & 0 \\ \mathbf{1} & 0 & \mathbf{1} \\ \mathbf{1} & \mathbf{1} & \mathbf{1} \end{bmatrix} \end{array} \qquad \begin{array}{l} \\ \texttt{incx} \\ \texttt{doublex} \\ \texttt{incy} \\ \texttt{incxmaybey} \end{array} \begin{array}{c} \begin{array}{ccc} \texttt{c} & \texttt{x} & \texttt{y} \end{array} \\ \begin{bmatrix} 0 & \mathbf{1} & 0 \\ 0 & \mathbf{1} & 0 \\ 0 & 0 & \mathbf{1} \\ 0 & \mathbf{1} & \mathbf{1} \end{bmatrix} \end{array}$$

(a) Read Matrix (b) May-Write Matrix

Fig. 3. Dependency matrices

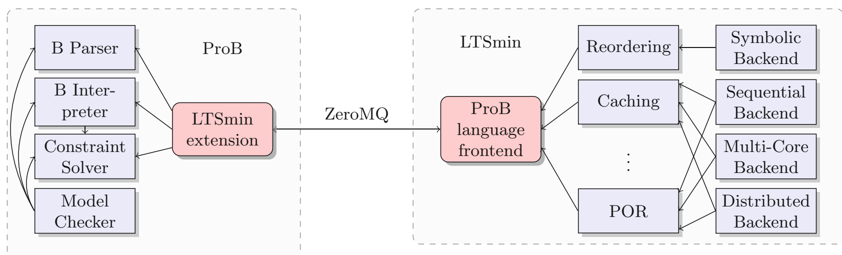


Fig. 4. Overview of the tool integration LTSmin↔ProB

2 Architecture Overview of LTSmin↔ProB Integration

LTSmin and ProB typically run as two separate processes which can be launched both manually or start each other. They are linked as shown in Fig. 4. The processes are linked via one IPC (local inter-process communication) socket per tool provided by ZeroMQ [18], a library offering distributed messaging. We employ a request-reply pattern on these sockets, where LTSmin sends requests and ProB responds. In order to handle messaging in ProB, we added a small layer in C.

In order to expose information about the loaded model to LTSmin, ProB's response to an initial request includes, amongst others, names of state variables and transition groups, an initial state and dependency matrices.

LTSmin expects a model to have a single initial state, while B and Event-B models allow for nondeterministic initialization. Thus, the initial state transferred to LTSmin consists of dummy values for all variables. Furthermore, we add a state variable named `is_init` that is initially set to false. Via a special transition `$init_state` which is only applicable if `is_init` is false, the actual initial states of the specification are exposed. For all these states (and their successors), `is_init` is set to true. This technicality leads to special cases in the entire implementation which we will omit.

In order to call ProB's next-state function, LTSmin sends a request containing the transition group and a state. ProB then will answer with a list of successor states. Since ProB is implemented in Prolog, it is hard to exchange states reasonably. Prolog terms have a limited life-span when using SICStus' foreign function interface. Thus, we serialize and deserialize state variables into/from blobs (binary large objects) by making use of an undocumented Prolog library

named `fastrw`. Each variable is stored in a separate blob, such that, a state is only a vector of blobs for LTSMIN. Naturally, repeated (de)serialization comes with an overhead that we chose to accept for now.

The labelling function is called in the same way, providing a label name and a state. PROB will answer with either true or false.

3 Implementation

In order to extend the prior integration, PROB needs to expose more of the B model to LTSMIN. In this section, we describe what information is additionally exchanged, how it is calculated and used by considering the running example in Fig. 2.

3.1 State Labels and Invariant Checking

In a labelled transition system, a set of atomic propositions is assumed. An atomic proposition is any predicate that we will call "state label". PROB will implement the labelling function, i.e., it will receive a state and a state label and return true or false.

The entire invariant can be seen as a single atomic proposition. However, if only some variables change from one state to its successor, not all conjuncts need to be re-evaluated. Thus, we split the invariant into its conjuncts. Each conjunct is announced as a state label to LTSMIN by providing a unique identifier. In order to expose which state label depends on which variable, additionally a state label matrix is included.

In our example in Fig. 2, initially, four state labels are created. The corresponding state label matrix is shown in Fig. 5.

$$
\begin{array}{r}
\\
x \in \mathbb{Z} \\
y \in \mathbb{Z} \\
x \le c \\
x + y \le 2c
\end{array}
\begin{array}{c}
\begin{array}{ccc} \texttt{c} & \texttt{x} & \texttt{y} \end{array} \\
\begin{bmatrix} 0 & \mathbf{1} & 0 \\ 0 & 0 & \mathbf{1} \\ \mathbf{1} & 0 & \mathbf{1} \\ \mathbf{1} & \mathbf{1} & \mathbf{1} \end{bmatrix}
\end{array}
$$

Fig. 5. State label dependency matrix

Since predicates are split at conjunctions, well-definedness issues might arise. As an example, consider the following predicate: $x \neq 0 \wedge 100 \ mod \ x = 1$. PROB's constraint solver will reorder the conjuncts in order to exclude any division by zero. Once the predicate is split into $x \neq 0$ and $100 \ mod \ x = 1$, it will only receive a single conjunct and cannot do any reordering. Thus, in a state with $x = 0$, the second conjunct on its own will result in a well-definedness error.

Thus, if a well-definedness error arose, another part of the original predicate has to be unsatisfied and we can assume the offending conjunct to be false as well.

Proof Information. Event-B models exported from Rodin [2] include information about discharged proof information. ProB uses this information, e.g., in order to avoid checking an invariant that has been proven to be preserved when a specific action is executed [5]. If an invariant has been fully proven to be correct, i.e., that it holds in the initial states and all transitions preserve it, we can exclude it from the list of invariants exposed to LTSmin.

Even though the machine in Fig. 2 is written in classical B, i.e., there is no proof information available, the two invariant conjuncts $x \in \mathbb{Z}$ and $y \in \mathbb{Z}$ are dropped. The type checker can already prove that they will hold. Thus, there is no need to check them separately.

3.2 Short States

In order to call ProB via an interface function, LTSmin needs to transmit the entire state to ProB. ProB then deserializes all variables individually before the interpreter is called. Obviously, not all values are required in order to calculate a state transition or evaluate a predicate, rendering some overhead obsolete. Such an interface function is called *long* function, analogously a state that consists of all state variables is called a *long* state.

Instead of transmitting the entire state, LTSmin can make use of the dependency matrix in order to *project* a long state to state vector that only contains the values of accessed variables. Such a state is named *short* state and is relative to either a state label or transition group. An interface function that receives a short state is, analogously, named a *short* function. Short states can also be *expanded* back to long states by inserting values for the missing variables, e.g., those of a predecessor or initial state.

Short states come in two flavors: firstly, *regular* short states are of a fixed size per transition group or state label and only contain values both written to and read from. Consider the initial state $init^{(c,x,y)} = (100, 0, 0)$ from Fig. 2. The corresponding short state for `incx` only contains c and x since y is not accessed. The projected short state, thus, is $init^{(c,x)}_{short} = (100, 0)$.

On the other hand, *R2W* short states (read to write) differ in the contained variables. LTSmin passes only read variables to ProB which in turn answers with written variables only. For `incx`, the R2W short state that is passed to ProB also is $init^{(c,x)}_{read} = (100, 0)$ because all written variables also are read. However, c is a constant, thus the value does not change. Thus, the returned state only consists of x, i.e., $init^{(c,x)}_{read} \rightarrow_1 = init^{(x)}_{write}$ with $init^{(x)}_{write} = (1)$.

Because there might be non-deterministic write accesses to variables ("may write"), a so-called *copy vector* is additionally passed to LTSmin. This copy vector is a bitfield marking which variables were actually written to and which values are taken from an earlier state. Additionally, the must-write matrix is given to LTSmin in order to expose which variables will be updated every single time.

The must-write matrix for Fig. 2 is given in Fig. 6. The difference to the may-write matrix is printed in bold. Note that if an entry in the must-write matrix

$$
\begin{array}{l}
 \\
\texttt{incx} \\
\texttt{doublex} \\
\texttt{incy} \\
\texttt{incxmaybey}
\end{array}
\begin{array}{c}
\begin{array}{ccc} \texttt{c} & \texttt{x} & \texttt{y} \end{array} \\
\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & \mathbf{0} \end{bmatrix}
\end{array}
$$

Fig. 6. Must-write matrix

is 1, it has to be 1 in the may-write matrix, but not vice versa. Then, consider the operation `incxmaybeincy` from Fig. 2. This operation only writes to y when x is an even number. Thus, for $s_{read}^{(c,x,y)} = (100, 2, 2)$ and $s \rightarrow_4 s'$, ${s'}_{write}^{(x,y)} = (3, 3)$ and $cpy_{s \rightarrow s'}^{(x,y)} = (0, 0)$, because both variables were actually written to. However, for $\hat{s}_{read}^{(c,x,y)} = (100, 3, 2)$ and $\hat{s} \rightarrow_4 \hat{s}'$, ${\hat{s}'}_{write}^{(x,y)} = (4, 2)$. Yet, $cpy_{\hat{s} \rightarrow \hat{s}'}^{(x,y)} = (0, 1)$ since y was not written to and the old value was copied.

Internally, LTSMIN may use both long and short states and convert freely between them. The PROB language frontend always communicates R2W short states in order to minimize overhead by communication and (de)serialization. However, the caching layer works on regular short states, while, e.g., the variable reordering uses long states.

3.3 Caching Mechanism

While the symbolic backend calls the next-state function once with a state that represents a set of states, the sequential backend calls it for each of the states and transition groups. Analogously, the same holds true for state labels and calls to the labelling function.

Since we already calculate dependency matrices, which contain information about which variables are read and, in case of the next-state function, are written to, we can calculate the corresponding short state instead. Then, in order to avoid transferring and (de)serializing states as well as calculating the same state transition or state label multiple times, we can store results in hash maps, one per transition group and state label each. These hash maps map the corresponding short state to either a list of (short) successors states or a Boolean value in case of state labels. Only if the lookup in the hash table fails, the state is transferred to PROB. Otherwise, LTSMIN can calculate the result by itself.

Currently, all operations are cached. Obviously, as more variables are accessed by an operation, caching offers lower benefit in exchange to the amount of memory consumed.

3.4 Guard Splitting

Operations (aka events or state transitions) are guarded, i.e., the action part that substitutes variables may only be executed if the guard predicate is satisfied. When LTSMIN asks PROB to calculate successor states for a given transition

group, ProB will evaluate the guard and, if applicable, try to find all (or a limited amount of) successors.

It is easy to make the following two observations: firstly, it is often more performant to evaluate single conjuncts of a guard individually. Usually, they access only a very limited amount of variables and can easily be cached. As an example, the guard $x < c$ of `incx` in Fig. 2 only reads two state variables (of which one is constant). Secondly, the same conjunct might guard multiple operations and, if evaluated for one operation in the same state, does not require additional evaluation for another operation. In Fig. 2, both `incx` and `incxmaybey` share the same guard $x < c$.

LTSmin's symbolic backend supports splitting the action from evaluating its guard. A new interface function next-action is provided that works similar to the next-state function, but assumes the guard of an operation to be true. Then, only the action part is evaluated. A special matrix (reads-action) is required for the next-action function that only contains variables that are read during the action part.

Additionally, each guard predicate is split into its conjuncts and associated with the corresponding transition groups in the guard matrix. Each conjunct is added to the state labels announced to LTSmin and their accessed variables are stored in the state label matrix. Parameter constraints however are considered to be part of the action. E.g., the guard $n > 0 \wedge n < c \wedge y < c$ of `incy(n)` in Fig. 2 is split into two: only $y < c$ is the actual guard for LTSmin and $n > 0 \wedge n < c$ is evaluated when calling the next-action function. The new matrices and the new rows in the state label matrix can be found in Fig. 7. Differences between the read matrix from Fig. 3 and the reads-action matrix are highlighted in bold.

$$\begin{array}{l} \\ \texttt{incx} \\ \texttt{doublex} \\ \texttt{incy} \\ \texttt{incxmaybey} \end{array} \begin{array}{c} \begin{array}{ccc} \texttt{c} & \texttt{x} & \texttt{y} \end{array} \\ \begin{bmatrix} \mathbf{0} & 1 & 0 \\ \mathbf{0} & 1 & 0 \\ \mathbf{0} & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \end{array}$$

(a) Reads-Action Matrix

$$\begin{array}{l} \\ \texttt{x} < \texttt{c} \\ \texttt{x} < \texttt{c}/2 \\ \texttt{x} > \texttt{0} \\ \texttt{y} < \texttt{c} \end{array} \begin{array}{c} \begin{array}{ccc} \texttt{c} & \texttt{x} & \texttt{y} \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \end{array}$$

(b) Extension of the State Label Matrix

$$\begin{array}{l} \\ \texttt{incx} \\ \texttt{doublex} \\ \texttt{incy} \\ \texttt{incxmaybey} \end{array} \begin{array}{c} \begin{array}{cccc} \texttt{x} < \texttt{c} & \texttt{x} < \texttt{c}/2 & \texttt{x} > \texttt{0} & \texttt{y} < \texttt{c} \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

(c) Guard Matrix

Fig. 7. Matrices for guard-splitting

3.5 Partial Order Reduction

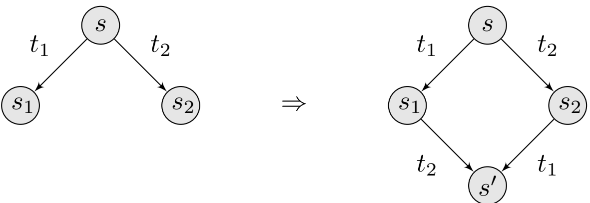
Partial order reduction (POR) [30,31] is a technique that reduces the amount of considered states based on a property that is checked. This is achieved by making use of additional information that can usually be inferred by static analysis.

In the following, we describe which relationships need to be calculated in order to achieve the best reduction with LTSMIN.

Definition 3 (According with, based on [24]). *Let $t_1, t_2 \in \mathcal{T}$ be any two operations. We define t_1 to be according with t_2, iff*

$$\forall s, s_1, s_2 \in \mathcal{R} : s \xrightarrow{t_1} s_1 \wedge s \xrightarrow{t_2} s_2 \implies \exists s' : s_1 \xrightarrow{t_2} s' \wedge s_2 \xrightarrow{t_1} s'$$

or as graphical representation:



We define that no $t \in \mathcal{T}$ accords with itself.

Accordance of transition groups expresses that they are independent from each other, i.e., depending on the property, not all interleavings have to be considered. ProB underapproximates the according-with relationship. Instead, the constraint $\forall s, s_1, s_2 \in \mathcal{S} : s \xrightarrow{t_i} s_1 \wedge s \xrightarrow{t_j} s_2 \implies \exists s' : s_1 \xrightarrow{t_j} s' \wedge s_2 \xrightarrow{t_i} s'$ is evaluated for a given pair $t_i, t_j \in \mathcal{T}, i \neq j$, considering the guards and the before-after predicates of both transitions. This does not ensure that any state is reachable. However, it is a valid *over*approximation of the does *not* accord relationship matrix that is passed to LTSMIN by negating all entries.

LTSMIN uses a heuristic in order to determine which of the calculated stubborn sets [30] might yield the best state space reduction. It requires a good approximation of the necessary enabling sets to do so:

Definition 4 (Necessary Enabling Set (NES), based on [24]). *Let g be any state label that is* ***disabled*** *in some state $s \in \mathcal{R}$, i.e. $\neg \mathbf{g}(\mathbf{s})$.*

A set of transitions $\mathcal{N}_g$ is called the necessary ***enabling*** *set for g in s, if for all states $s' \in \mathcal{R}$ with some sequence $s \xrightarrow{t_1,\ldots,t_n} s'$ and $\mathbf{g}(\mathbf{s'})$, for at least one transition t_i $(1 \leq i \leq n)$ we have $t_i \in \mathcal{N}_g$.*

We can use an already existing implementation of the `test_path` procedure (cf. [14], definition 2) in order to calculate the necessary enabling set. In the implementation, it is just tested for any given state, whether a single transition can enable the guard label. In particular, this means that the states s and s' in Definition 4 may not be reachable at all. This results in a safe approximation but may lose precision.

Along with the NES, a necessary *disabling* set (NDS) is approximated. It is calculated in the same way but uses the negation of the state label.

Information about the NES and NDS matrices can syntactically be approximated from the dependency matrix. Solving the given constraints often results in a better approximation and, thus, a better reduction.

Furthermore, with additional information one might prove that out of at least three transition, e.g., t_1, t_2 and t', some transitions, e.g., t_2 and t' might not be enabled at the same time. Then, not all interleaving of t_1 and t_2 need to be considered. This situation is depicted in Fig. 8, where s_1 does not have to be visited. Then, a may-be co-enabled matrix is calculated, based on the following definition:

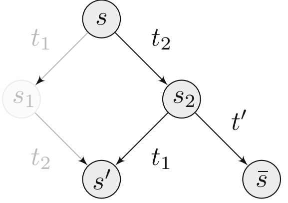


Fig. 8. Reduction using co-enabledness

Definition 5 (Co-Enabledness, based on [24]). *Two state labels $l, l' \in \mathcal{L}$ are co-enabled in a state $s \in \mathcal{R}$ iff they both evaluate to true in s, i.e. $l(s) = true = l'(s)$.*

Again, instead of working on reachable states, it is checked whether there is *any* state in the Cartesian product of types where both labels are enabled. If the co-enabledness of two state labels cannot be determined, they are considered as may-be co-enabled.

While ProB offers an implementation partial order reduction as well [15,16], the partial order reduction algorithm implemented in LTSmin uses a finer heuristic. ProB checks whether a transition can enable another transition, LTSmin uses information about whether individual guards of the event can be enabled, often resulting in a better reduction.

However, this reduction comes with a tradeoff: to calculate the additional matrices, more constraints have to be solved by ProB in order to determine the additional relationships, resulting in a longer analysis time. ProB only calculates accordance of transitions and one row in the NES matrix per transition instead of per guard conjunct. Furthermore, LTSmin does not allow both transitions to be enabled and not generating any successors at the same time. This is possible in ProB when no suitable parameters exist. Thus, an additional guard is added which is an existential quantification of the parameters and often rather costly to evaluate. This quantification has to be solved both on evaluation of the guard and computation of successor states.

3.6 LTL Formula Checking

In order to check LTL properties, both tools need to have access to the formula: only ProB is capable of dealing with atomic propositions properly since it implements the syntax and semantics of the B language. LTSmin, however, requires the formula in order to generate the corresponding Büchi automaton.

Thus, ProB parses the formula first. All atomic propositions in the formula are replaced with a newly generated state label. In order to evaluate these new atomic propositions, the state labelling function is extended in ProB. Furthermore, the formula is wrapped in a “next” operator in order to skip the artificial

initial state introduced earlier. This modified formula is then pretty printed into a format that LTSMIN can parse.

4 Evaluation

In this section, we will evaluate the performance of the tool integration of LTSMIN and PROB using both the sequential and symbolic backend. We will compare the model checking time on several models from literature and industrial applications which are publicly available under https://github.com/pkoerner/prob-ltsmin-models. Benchmark scripts are included in the repositories as well.

Furthermore, we will compare the impact of the implementations of partial order reduction for a different set of models, where the state space can be reduced.

Each benchmark was run on a machine featuring two Intel Xeon IvyBridge E5-2697 with twelve cores each running at 2.70 GHz and 100 GB of RAM. Two CPUs were reserved for each run of invariant verification, partial order reduction and LTL model checking. For multi-core benchmarks, we reserved as many CPUs as there are worker threads plus one CPU for ZeroMQ overhead.

The given values are the median value of ten repetitions.

4.1 Invariant Checking

We benchmarked three backends on multiple B and Event-B models:

- PROB: the vanilla PROB model checker
- LTSMIN (seq): the sequential backend of LTSMIN with the PROB interpreter,
- LTSMIN (sym): the symbolic backend of LTSMIN without guard-splitting and the PROB interpreter.

We omit results with guard-splitting enabled since they are very similar to the symbolic backend without guard-splitting for applicable models, i.e., those written in classical B.

Runtimes and memory consumption can be found in Table 1. Runtimes of LTSMIN's sequential and symbolic backend do not include PROB's startup time which includes parsing and minor analysis of the model. None of the considered models has any invariant violation and, thus, all models have to be explored exhaustively.

Only for one of models benchmarked, the "Set Laws" machine, a single backend of LTSMIN is slower than PROB. Apart from that, we can observe speed-ups ranging from two-fold up to more than two hundred times. For most models, LTSMIN is at least an order of magnitude faster than PROB.

The "Train" machine cannot be checked by vanilla PROB on the benchmarking machine, as it runs out of main memory after three days, exploring about half the state space. Both LTSMIN backends manage to verify the entire state space in several hours. Only in few instances, LTSMIN requires more memory

Table 1. Invariant checking performance (runtime in seconds, memory in MB) of ProB alone compared to LTSMin with ProB. †: Estimated runtime, ‡: limited amount of initializations

	Tool	ProB	LTSMin (seq)	LTSMin (sym)
Four Slot	Runtime	26.33	0.99	1.12
(Simpson's Algorithm)	Speed-up	1.00	26.60	23.51
46657 states	Memory	227.21	11.14	426.07
Landing Gear	Runtime	61.38	1.04	0.65
[17]	Speed-up	1.00	59.02	94.43
43306 states	Memory	244.01	11.95	425.77
RETHER protocol	Runtime	77.75	4.87	6.09
[33]	Speed-up	1.00	15.97	12.77
42253 states	Memory	304.08	12.61	430.36
Set Laws	Runtime	232.37	120.57	301.03
	Speed-up	1.00	1.93	0.77
35937 states	Memory	428.45	87.05	501.86
Earley Parser	Runtime	24612.00	15153.00	6476.00
(J.-R. Abrial)	Speed-up	1.00	1.62	3.80
472886 states	Memory	4218.62	5224.57	4833.13
CAN Bus	Runtime	131.51	2.68	2.80
(John Colley)	Speed-up	1.00	49.07	46.97
132599 states	Memory	346.74	24.14	435.50
Mercury Orbiter	Runtime	2608.28	14.14	10.76
[13]	Speed-up	1.00	184.46	242.41
589278 states	Memory	2360.06	68.66	428.34
Mode Protocol	Runtime	1393.97	317.90	381.20
‡ [13]	Speed-up	1.00	4.38	3.66
336648 states	Memory	1097.70	151.36	536.55
Core	Runtime	1921.58	315.77	320.05
	Speed-up	1.00	6.09	6.00
160946 states	Memory	1751.64	314.94	742.17
Train	Runtime	600000 †	33124.00	49120.00
[1]	Speed-up	1.00	18.11	12.21
61648077 states	Memory	>100000	18887.32	19815.79
Train	Runtime	98.59	51.79	71.01
(reduced version)	Speed-up	1.00	1.90	1.39
24636 states	Memory	198.48	39.00	493.39

than ProB. Surprisingly, the sequential backend often requires an order of magnitude less memory, even though it maintains a cache. In the only instance where it uses more memory, i.e., "Earley Parser", only few variables re-use the same values. Thus, almost no sharing between in states is possible and the entire fastrw representation for almost every state has to be stored.

Overall, LTSmin is able to outperform ProB in almost all instances. Obviously, caching is really important for the sequential backend. We tried two implementations of a similar caching mechanism for ProB in order to benefit from similar speed-ups: firstly, by hashing short states as well as asserting them as Prolog facts, and, secondly, by serializing the state via the fastrw library and storing the result in a hash map in C. However, neither implementation had a similar impact. Often, they even slowed ProB down. In the first case, we assume that the hashing algorithms for Prolog terms are too slow. For the second implementation, the overhead to serialize every state, in particular for cache lookups, was too costly.

4.2 LTL Model Checking

We benchmarked LTL model checking on a few of aforementioned models that are reasonably sized. Since no LTL formulas were included in the models, we arbitrarily picked some that allow reasoning about the models. ProB's special syntax is used in the formulas: $e(x)$ means that the operation x is enabled. The results are given in Table 2.

We can see that for LTL formulas that hold, the good speed-ups and low memory consumption of LTSmin that was presented in Sect. 4.1 can also be observed for LTL model checking. If a formula is not satisfied, LTSmin can be more than thousandfold faster. While ProB generates the state space of the transition system first, LTSmin features an on-the-fly LTL model checking algorithm where the state space consisting of the Cartesian product of the corresponding Büchi automaton and the actual transition system is generated as necessary. Thus, accepting loops can be found quickly without exploring the entire transition system and speed-ups may be more than thousandfold.

4.3 Partial Order Reduction

From benchmarks conducted in [16,20], it became clear that, in typical B and Event-B models, partial order reduction usually does not work well in combination with invariant verification. For the models above, none or very little reduction was achievable. Instead, we compare the results of partial order reduction for deadlock checking.

In ProB, we use the "least" heuristic for the partial order reduction. In order to reduce analysis time for LTSmin, the timeout is set to 20 ms per predicate.

Table 2. Runtimes (in seconds) and speed-ups of LTL model checking. †: LTL formula does not hold. Speed-up compared to ProB without LTSmin.

	Tool	ProB	LTSmin
RETHER protocol [33] $G\ (not\ e(reserve)$ $\implies X(e(grant)\ \&\ e(no_grant)))$ †	Runtime	76.67	0.14
	Speed-up	1.00	547.64
	Memory	335.92	5.57
RETHER protocol [33] $GF\ e(pass_token)$	Runtime	77.50	4.79
	Speed-up	1.00	16.18
	Memory	335.31	15.02
CAN Bus (John Colley) $GF\ e(Update)$	Runtime	125.49	3.18
	Speed-up	1.00	39.46
	Memory	439.01	29.21
CAN Bus (John Colley) $G\ (e(T1Wait) \implies X(T1_timer > 0))$ †	Runtime	126.80	0.24
	Speed-up	1.00	528.33
	Memory	435.25	6.23
Mode Protocol [13] $GF\ e(evt_DeliverOK)$ †	Runtime	1406.91	0.30
	Speed-up	1.00	4689.70
	Memory	1336.25	6.39

Results are shown in Table 3 comparing the performance of ProB's POR for deadlock checking with the one of LTSmin. B models that offer no reduction with either tool are omitted (which are more than half). Again, the startup time of ProB that includes parsing the machine file, is not included.

As can be seen in Table 3, LTSmin is – as discussed in Sect. 3.5 – able to find a reduction that is equal to ProB's or even better (for the "Set Laws" machine, the additional state is the artificial initial state). Indeed, for the "Mercury Orbiter" and "Landing Gear", ProB cannot reduce the state space at all with the given heuristic, whereas LTSmin reduces the state space by about a quarter up to a half.

However, fine-tuning of the time-outs for the static analysis is important. For machines with many unique guards, analysis time can easily exceed model checking time. This can be observed for the "Landing Gear" and the extremely reduced "Four Slot". With the default time-out value of 300 ms, the analysis time of the "Mercury Orbiter" can exceed a minute, which is more than four times the time required for model checking without any reduction. A reason could be that the constraint solver cannot solve the necessary predicates easily and time-outs are raised often.

Table 3. Runtimes (in seconds) and impact of POR in ProB alone and LTSmin with ProB for deadlock checking.

	Tool	ProB	LTSmin
Four Slot 46657 states	Analysis time	0.08	0.23
	Model checking time	24.37	0.90
	States (after reduction)	44065	44065
Landing Gear [17] 43306 states	Analysis Time	0.98	2.43
	Model checking time	87.94	1.25
	States (after reduction)	43306	29751
Set Laws 35937 states	Analysis time	0.30	0.37
	Model checking time	0.14	0.07
	States (after reduction)	33	34
CAN Bus (John Colley) 132599 states	Analysis time	1.38	1.49
	Model checking time	94.90	2.24
	States (after reduction)	85515	67006
Mercury Orbiter [13] 589278 states	Analysis time	51.00	9.29
	Model checking time	2757.33	16.19
	States (after reduction)	589278	316164

4.4 Multi-core Model Checking

In order to evaluate the performance of the multi-core backend, we performed multi-core invariant verification on the five B models with the largest runtime in Sect. 4.1.

The runtime of runs with different amount of workers are shown in Table 4. Speed-ups are visualized in Fig. 9. This time, ProB's startup time is included because the multi-core extension of LTSmin starts up ProB.

Currently, each worker thread maintains its own cache. Since caching works fairly well for many B machines, it is quite costly to fill each cache individually. This explains that often, for the first few workers the speed-up is nearly linear, but grows slower when more than ten workers participate.

An exception is the "Earley Parser": the standalone sequential backend does not offer much speed-up (cf. Sect. 4.1). Thus, the caching effects are lower. Additionally, ProB spends much time deserializing the state variables because the Prolog terms grow quite large. Hence, a more linear speed-up is possible for many workers.

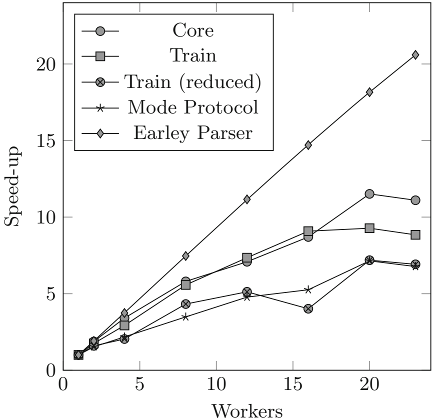


Fig. 9. Speed-ups of multi-core LTSMIN+PROB model checking for complex models

Table 4. Runtimes (in seconds) and speed-ups of multi-core LTSMIN with PROB model checking on the high-performance cluster. ‡: Limited amount of initializations

Earley Parser	Workers	1	4	8	12	16	20	23
(J.-R. Abrial)	Runtime	15152	4051	2030.76	1358.06	1030.59	834.46	735.76
472886 states	Speed-up	1.00	3.74	7.46	11.16	14.70	18.16	20.59
Mode Protocol	Workers	1	4	8	12	16	20	23
‡ [13]	Runtime	328.29	150.71	94.23	68.74	62.46	45.96	48.48
336648 states	Speed-up	1.00	2.18	3.48	4.78	5.26	7.14	6.77
Core	Workers	1	4	8	12	16	20	23
	Runtime	317.48	93.18	54.73	44.82	36.48	27.57	28.60
160946 states	Speed-up	1.00	3.41	5.80	7.08	8.70	11.52	11.10
Train	Workers	1	4	8	12	16	20	23
[1]	Runtime	32805	11215	5889	4464	3612	3536.60	3710
61648077 states	Speed-up	1.00	2.93	5.57	7.35	9.08	9.28	8.84
Train	Workers	1	4	8	12	16	20	23
(reduced)	Runtime	62.39	30.59	14.43	12.18	15.54	8.68	9.02
24636 states	Speed-up	1.00	2.04	4.32	5.12	4.01	7.19	6.92

5 Conclusion, Related and Future Work

In this paper, we presented and evaluated significant improvements of the existing link of PROB and LTSMIN. It allows state-of-the-art model checking of industrial-sized models with large state spaces, where the vanilla PROB model checker struggles due to time or memory constraints. E.g., the “Train” example can only be checked successfully with PROB on its own by distributing the workload onto several machines, whereas with LTSMIN, it could be verified on an ordinary notebook or workstation.

We have compared symbolic reachability analysis to the impact of alternative techniques that can speed up state space generation, e.g., partial order reduction and symmetry reduction in [4]. A discussion of the impact of algorithms that are implemented in both LTSmin and ProB can also be found in Sect. 4.

Additionally, there is another toolset named `libits` [12]. It supports, like LTSmin, symbolic model checking using decision diagrams, variable reordering and LTL as well as CTL model checking. As we understand, its input formalisms are translated into its guarded action language (GAL). An integration for B might prove to be infeasible because a constraint solver is required in order to compute some state transitions and would have to be implemented in GAL itself. We do not know yet whether linking ProB with `libits` in order to compute state transitions is possible.

The caching performed by LTSmin seems to be particularly efficient. For several realistic examples, the ProB and LTSmin link achieves two to three orders of magnitude improvements in runtime. Yet, there are several aspects that require future work: currently, the ProB front-end of LTSmin does not support parallel symbolic model checking with Sylvan [32]. Furthermore, caches are local per worker. A shared cache will most likely improve the scaling for massive parallel model checking. Additionally, the cache does not implement R2W semantics, which loses information about write accesses and requires more memory and additional state transformations.

Moreover, there is room for interesting research: while we know from experience that, for most B models, partial order reduction often only has little to no impact on the state space, we are unsure why. Would alternative algorithms perform better? Is the constraint solver not strong enough? Is the approximation given to LTSmin not precise enough? Does partial order reduction not perform with the modeling style employed in B? If so, are there any patterns which hinder it? Additionally, a proper survey of distributed model checking of B and Event-B specifications – which we did not touch upon due to page limitations – between, e.g. LTSmin's distributed backend, ProB's *distb* [21] and TLC [35] should be considered.

With the gained experience and shared know-how about both LTSmin and ProB, we now aim to extend the implementation for CSP||B, where the execution of classical B machine is guided by a CSP specification. While ProB provides both an animator and model checker for this formalism, ProB currently is not a satisfactory tool for this formalism. Interleaving of several processes causes an enormous state space explosion that we hope the symbolic capabilities of LTSmin can manage.

Acknowledgement. Computational support and infrastructure was provided by the "Centre for Information and Media Technology" (ZIM) at the University of Düsseldorf (Germany). We also thank Ivaylo Dobrikov and Alfons Laarman for their helpful explanations concerning partial order reduction algorithms.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering, 1st edn. Cambridge University Press, Cambridge (2010)
2. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int. J. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
3. Abrial, J.-R., Lee, M.K.O., Neilson, D.S., Scharbach, P.N., Sørensen, I.H.: The B-method. In: Prehn, S., Toetenel, H. (eds.) VDM 1991. LNCS, vol. 552, pp. 398–405. Springer, Heidelberg (1991). https://doi.org/10.1007/BFb0020001
4. Bendisposto, J.: Symbolic reachability analysis of B through ProB and LTSmin. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 275–291. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_18
5. Bendisposto, J., Leuschel, M.: Proof assisted model checking for B. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 504–520. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10373-5_26
6. Bicarregui, J.C., Fitzgerald, J.S., Larsen, P.G., Woodcock, J.C.P.: Industrial practice in formal methods: a review. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 810–813. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05089-3_52
7. Blom, S., van de Pol, J.: Symbolic reachability for process algebras with recursive data types. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 81–95. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85762-4_6
8. Blom, S., van de Pol, J., Weber, M.: LTSmin: distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_31
9. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.-J.: Symbolic model checking: 1020 states and beyond. Inf. Comput. **98**(2), 142–170 (1992)
10. Carlsson, M., et al.: SICStus Prolog user's manual. Swedish Institute of Computer Science Kista (1988)
11. Ciardo, G., Marmorstein, R.M., Siminiceanu, R.: The saturation algorithm for symbolic state-space exploration. STTT **8**(1), 4–25 (2006)
12. Colange, M., Baarir, S., Kordon, F., Thierry-Mieg, Y.: Towards distributed software model-checking using decision diagrams. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 830–845. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_58
13. DEPLOY Deliverable D20: Report on Pilot Deployment in the Space Sector. FP7 ICT DEPLOY Project, January 2010. http://www.deploy-project.eu/html/deliverables.html
14. Dobrikov, I., Leuschel, M.: Optimising the ProB model checker for B using partial order reduction. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 220–234. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10431-7_16
15. Dobrikov, I., Leuschel, M.: Optimising the ProB model checker for B using partial order reduction. Form. Asp. Comput. **28**(2), 295–323 (2016)
16. Dobrikov, I.M.: Improving explicit-state model checking for B and Event-B. Ph.D. thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf (2017)

17. Hansen, D., Ladenberger, L., Wiegard, H., Bendisposto, J., Leuschel, M.: Validation of the ABZ landing gear system using ProB. In: Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. CCIS, vol. 433, pp. 66–79. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07512-9_5
18. Hintjens, P.: ZeroMQ: Messaging for Many Applications. O'Reilly Media Inc., Sebastopol (2013)
19. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
20. Körner, P.: An integration of ProB and LTSmin. Master's thesis, Heinrich Heine Universität Düsseldorf, February 2017
21. Körner, P., Bendisposto, J.: Distributed model checking using ProB. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) NFM 2018. LNCS, vol. 10811, pp. 244–260. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77935-5_18
22. Krings, S., Leuschel, M.: Proof assisted symbolic model checking for B and Event-B. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 135–150. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_8
23. Krings, S., Leuschel, M.: SMT solvers for validation of B and Event-B models. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 361–375. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_23
24. Laarman, A., Pater, E., van de Pol, J., Weber, M.: Guard-based partial-order reduction. In: Bartocci, E., Ramakrishnan, C.R. (eds.) SPIN 2013. LNCS, vol. 7976, pp. 227–245. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39176-7_15
25. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_46
26. Meijer, J., Kant, G., Blom, S., van de Pol, J.: Read, write and copy dependencies for symbolic model checking. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 204–219. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13338-6_16
27. Meijer, J., van de Pol, J.: Bandwidth and wavefront reduction for static variable ordering in symbolic reachability analysis. In: Rayadurgam, S., Tkachuk, O. (eds.) NFM 2016. LNCS, vol. 9690, pp. 255–271. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40648-0_20
28. Plagge, D., Leuschel, M.: Validating B, Z and TLA$^+$ Using ProB and Kodkod. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 372–386. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_31
29. Spermann, C., Leuschel, M.: ProB gets nauty: effective symmetry reduction for B and Z models. In: Proceedings TASE, pp. 15–22. IEEE (2008)
30. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) ICATPN 1989. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-53863-1_36
31. Valmari, A.: A stubborn attack on state explosion. In: Clarke, E.M., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 156–165. Springer, Heidelberg (1991). https://doi.org/10.1007/BFb0023729
32. van Dijk, T., van de Pol, J.: Sylvan: multi-core framework for decision diagrams. STTT **19**(6), 675–696 (2017)

33. Venkatramani, C., Chiueh, T.-C.: Design, implementation, and evaluation of a software-based real-time ethernet protocol. ACM SIGCOMM Comput. Commun. Rev. **25**(4), 27–37 (1995)
34. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: practice and experience. ACM Comput. Surv. (CSUR) **41**(4), 19:1–19:36 (2009)
35. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_6

Towards a Formal Notion of Impact Metric for Cyber-Physical Attacks

Ruggero Lanotte[1], Massimo Merro[2](✉), and Simone Tini[1]

[1] Dipartimento di Scienza e Alta Tecnologia, Università dell'Insubria, Como, Italy
{ruggero.lanotte,simone.tini}@uninsubria.it

[2] Dipartimento di Informatica, Università degli Studi di Verona, Verona, Italy
massimo.merro@univr.it

Abstract. Industrial facilities and critical infrastructures are transforming into "smart" environments that dynamically adapt to external events. The result is an ecosystem of heterogeneous physical and cyber components integrated in cyber-physical systems which are more and more exposed to *cyber-physical attacks*, *i.e.*, security breaches in cyberspace that adversely affect the physical processes at the core of the systems.

We provide a formal *compositional metric* to estimate the *impact* of cyber-physical attacks targeting sensor devices of *IoT systems* formalised in a simple extension of Hennessy and Regan's *Timed Process Language*. Our *impact metric* relies on a discrete-time generalisation of Desharnais et al.'s *weak bisimulation metric* for concurrent systems. We show the adequacy of our definition on two different attacks on a simple surveillance system.

Keywords: IoT system · Cyber-physical attack · Impact metric
Probabilistic metric semantics

1 Introduction

The *Internet of Things* (IoT) is heavily affecting our daily lives in many domains, ranging from tiny wearable devices to large industrial systems with thousands of heterogeneous cyber and physical components that interact with each other.

Cyber-Physical Systems (CPSs) are integrations of networking and distributed computing systems with physical processes, where feedback loops allow the latter to affect the computations of the former and vice versa. Historically, CPSs relied on proprietary technologies and were implemented as stand-alone networks in physically protected locations. However, the growing connectivity and integration of these systems has triggered a dramatic increase in the number

Partially supported by the project "Dipartimenti di Eccellenza 2018–2022", funded by the Italian Ministry of Education, Universities and Research (MIUR), and by the Joint Project 2017 "Security Static Analysis for Android Things", jointly funded by the University of Verona and JuliaSoft Srl.

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 296–315, 2018.
https://doi.org/10.1007/978-3-319-98938-9_17

of *cyber-physical attacks* [14], *i.e.*, security breaches in cyberspace that adversely affect the physical processes, *e.g.*, manipulating *sensor readings* and, in general, influencing physical processes to bring the system into a state desired by the attacker.

Cyber-physical attacks are complex and challenging as they usually cross the boundary between cyberspace and the physical world, possibly more than once [11]. Some notorious examples are: (i) the *STUXnet* worm, which reprogrammed PLCs of nuclear centrifuges in Iran [6], (ii) the attack on a sewage treatment facility in Queensland, Australia, which manipulated the SCADA system to release raw sewage into local rivers [32], or the (iii) the recent *BlackEnergy* cyber-attack on the Ukrainian power grid, again compromising the SCADA system [15].

The points in common of these systems is that they are all safety critical and failures may cause catastrophic consequences. Thus, the concern for consequences at the physical level puts *CPS security* apart from standard *IT security.*

Timing is particularly relevant in CPS security because the physical state of a system changes continuously over time and, as the system evolves in time, some states might be more vulnerable to attacks than others. For example, an attack launched when the target state variable reaches a local maximum (or minimum) may have a great impact on the whole system behaviour [17]. Also the *duration of the attack* is an important parameter to be taken into consideration in order to achieve a successful attack. For example, it may take minutes for a chemical reactor to rupture, hours to heat a tank of water or burn out a motor, and days to destroy centrifuges.

The estimation of the *impact* of cyber-physical attacks on physical components of the target system is a crucial task when protecting CPSs [10]. For instance, in industrial CPSs, before taking any countermeasure against an attack, engineers first try to estimate the impact of the attack on the system functioning (*e.g.*, performance and security) and weight it against the cost of stopping the plant. If this cost is higher than the damage caused by the attack (as is sometimes the case), then engineers might actually decide to let the system continue its activities even under attack. Thus, once an attack is detected, *impact metrics* are necessary to quantify the perturbation introduced in the physical behaviour of the system under attack.

The *goal* of this paper is to lay theoretical foundations to provide formal instruments to precisely define the notion of impact of cyber-physical attack targeting physical devices, such as *sensor devices* of IoT systems. For that we rely on a timed generalisation of *weak bisimulation metrics* [5] to compare the behaviour of two systems up to a given tolerance, for time-bounded executions.

Weak bisimulation metric allows us to compare two systems M and N, writing $M \simeq_p N$, if the weak bisimilarity holds with a *distance* or *tolerance* $p \in [0,1]$, *i.e.*, if M and N exhibit a different behaviour with probability p, and the same behaviour with probability $1-p$. A useful generalisation is the *n-bisimulation metric* [3] that takes into account bounded computations. Intuitively, the distance p is ensured only for the first n computational steps, for some $n \in \mathbb{N}$.

However, in timed systems it is desirable to focus on the passage of time rather than the number of computational steps. This would allow us to deal with situations where it is not necessary (or it simply does not make sense) to compare two systems "ad infinitum" but only for a limited amount of time.

Contribution. In this paper, we first introduce a general notion of *timed bisimulation metric* for concurrent probabilistic systems equipped with a discrete notion of time. Intuitively, this kind of metric allows us to derive a *timed weak bisimulation with tolerance*, denoted with $\approx_p^k$, for $k \in \mathbb{N}^+ \cup \{\infty\}$ and $p \in [0,1]$, to express that the tolerance p between two timed systems is ensured only for the first k time instants (tick-actions). Then, we use our timed bisimulation metric to set up a formal *compositional* theory to study and measure the *impact* of cyber-physical attacks on IoT systems specified in a simple probabilistic timed process calculus which extends Hennessy and Regan's *Timed Process Language* (TPL) [12]. IoT systems in our calculus are modelled by specifying: (i) a *physical environment*, containing informations on the physical state variables and the sensor measurements, and (ii) a *logics* that governs both accesses to sensors and channel-based communications with other cyber components.

We focus on *attacks on sensors* that may eavesdrop and possibly modify the sensor measurements provided to the controllers of sensors, affecting both the *integrity* and the *availability* of the system under attack.

In order to make security assessments of our IoT systems, we adapt a well-know approach called *Generalized Non Deducibility on Composition* (GNDC) [7] to compare the behaviour of an IoT system M with the behaviour of the same system under attack, written $M \parallel A$, for some arbitrary cyber-physical attack A. This comparison makes use of our timed bisimulation metric to evaluate not only the *tolerance* and the *vulnerability* of a system M with respect to a certain attack A, but also the *impact* of a successful attack in terms of the deviation introduced in the behaviour of the target system. In particular, we say that a system M *tolerates an attack* A if $M \parallel A \approx_0^\infty M$, *i.e.*, the presence of A does not affect the behaviour of M; whereas M is said to be *vulnerable* to A in the time interval $m..n$ with impact p if $m..n$ is the smallest interval such that $M \parallel A \approx_0^{m-1} M$ and $M \parallel A \approx_p^k M$, for any $k \geq n$, *i.e.*, if the perturbation introduced by the attack A becomes observable in the m-th time slot and yields the maximum *impact* p in the n-th time slot. In the concluding discussion we will show that the *temporal vulnerability window* $m..n$ provides several informations about the corresponding attack, such as *stealthiness* capability, duration of the *physical effects* of the attack, and consequent room for possible run-time *countermeasures*.

As a case study, we use our timed bisimulation metric to measure the impact of two different attacks injecting *false positives* and *false negative*, respectively, into a simple surveillance system expressed in our process calculus.

Outline. Section 2 formalises our timed bisimulation metrics in a general setting. Section 3 provides a simple calculus of IoT systems. Section 4 defines cyber-physical attacks together with the notions of tolerance and vulnerability *w.r.t.*

an attack. In Sect. 5 we use our metrics to evaluate the impact of two attacks on a simple surveillance system. Section 6 draws conclusions and discusses related and future work. In this extended abstract proofs are omitted, full details of the proofs can be found in the technical report [23].

2 Timed Bisimulation Metrics

In this section, we introduce *timed bisimulation metrics* as a general instrument to derive a notion of timed and approximate weak bisimulation between probabilistic systems equipped with a discrete notion of time. In Sect. 2.1, we recall the semantic model of *nondeterministic probabilistic labelled transition systems*; in Sect. 2.2, we present our metric semantics.

2.1 Nondeterministic Probabilistic Labelled Transition Systems

Nondeterministic probabilistic labelled transition systems (pLTS) [30] combine classic LTSs [16] and discrete-time Markov chains [34] to model, at the same time, reactive behaviour, nondeterminism and probability. We first provide the mathematical machinery required to define a pLTS.

The state space in a pLTS is given by a set $\mathcal{T}$, whose elements are called *processes*, or *terms*. We use $t, t', ..$ to range over $\mathcal{T}$. A (discrete) *probability sub-distribution* over $\mathcal{T}$ is a mapping $\Delta\colon \mathcal{T} \to [0,1]$, with $\sum_{t\in\mathcal{T}} \Delta(t) \in (0,1]$. We denote $\sum_{t\in\mathcal{T}} \Delta(t)$ by $|\,\Delta\,|$, and we say that Δ is a *probability distribution* if $|\Delta|= 1$. The *support* of Δ is given by $\lceil\Delta\rceil = \{t \in \mathcal{T} : \Delta(t) > 0\}$. The set of all sub-distributions (resp. distributions) over $\mathcal{T}$ with finite support will be denoted with $\mathcal{D}_{\rm sub}(\mathcal{T})$ (resp. $\mathcal{D}(\mathcal{T})$). We use Δ, Θ, Φ to range over $\mathcal{D}_{\rm sub}(\mathcal{T})$ and $\mathcal{D}(\mathcal{T})$.

Definition 1 (pLTS [30]**).** *A* pLTS *is a triple* $(\mathcal{T}, \mathbf{A}, \rightarrow)$*, where: (i)* $\mathcal{T}$ *is a countable set of* terms*, (ii)* $\mathbf{A}$ *is a countable set of* actions*, and (iii)* $\rightarrow \subseteq \mathcal{T} \times \mathbf{A} \times \mathcal{D}(\mathcal{T})$ *is a* transition relation*.*

In Definition 1, we assume the presence of a special deadlocked term $\mathsf{Dead} \in \mathcal{T}$. Furthermore, we assume that the set of actions $\mathbf{A}$ contains at least two actions: τ and tick. The former to model internal computations that cannot be externally observed, while the latter denotes the passage of one time unit in a setting with a discrete notion of time [12]. In particular, tick is the only *timed action* in $\mathbf{A}$.

We write $t \xrightarrow{\alpha} \Delta$ for $(t, \alpha, \Delta) \in \rightarrow$, $t \xrightarrow{\alpha}$ if there is a distribution $\Delta \in \mathcal{D}(\mathcal{T})$ with $t \xrightarrow{\alpha} \Delta$, and $t \not\xrightarrow{\alpha}$ otherwise. Let $der(t, \alpha) = \{\Delta \in \mathcal{D}(\mathcal{T}) \mid t \xrightarrow{\alpha} \Delta\}$ denote the set of the derivatives (i.e. distributions) reachable from term t through action α. We say that a pLTS is *image-finite* if $der(t, \alpha)$ is finite for all $t \in \mathcal{T}$ and $\alpha \in \mathbf{A}$. In this paper, we will always work with image-finite pLTSs.

Weak Transitions. As we are interested in developing a *weak* bisimulation metric, we need a definition of weak transition which abstracts away from τ-actions. In a probabilistic setting, the definition of weak transition is somewhat complicated

by the fact that (strong) transitions take terms to distributions; consequently if we are to use weak transitions then we need to generalise transitions, so that they take (sub-)distributions to (sub-)distributions.

To this end, we need some extra notation on distributions. For a term $t \in \mathcal{T}$, the *point (Dirac) distribution at* t, denoted $\overline{t}$, is defined by $\overline{t}(t) = 1$ and $\overline{t}(t') = 0$ for all $t' \neq t$. Then, the convex combination $\sum_{i \in I} p_i \cdot \Delta_i$ of a family $\{\Delta_i\}_{i \in I}$ of (sub-)distributions, with I a finite set of indexes, $p_i \in (0, 1]$ and $\sum_{i \in I} p_i \leq 1$, is the (sub-)distribution defined by $(\sum_{i \in I} p_i \cdot \Delta_i)(t) \stackrel{def}{=} \sum_{i \in I} p_i \cdot \Delta_i(t)$ for all $t \in \mathcal{T}$. We write $\sum_{i \in I} p_i \cdot \Delta_i$ as $p_1 \cdot \Delta_1 + \ldots + p_n \cdot \Delta_n$ when $I = \{1, \ldots, n\}$.

Thus, we write $t \xrightarrow{\hat{\tau}} \Delta$, for some term t and some distribution Δ, if either $t \xrightarrow{\tau} \Delta$ or $\Delta = \overline{t}$. Then, for $\alpha \neq \tau$, we write $t \xrightarrow{\hat{\alpha}} \Delta$ if $t \xrightarrow{\alpha} \Delta$. Relation $\xrightarrow{\hat{\alpha}}$ is extended to model transitions from sub-distributions to sub-distributions. For a sub-distribution $\Delta = \sum_{i \in I} p_i \cdot \overline{t_i}$, we write $\Delta \xrightarrow{\hat{\alpha}} \Theta$ if there is a non-empty set of indexes $J \subseteq I$ such that: (i) $t_j \xrightarrow{\hat{\alpha}} \Theta_j$ for all $j \in J$, (ii) $t_i \not\xrightarrow{\hat{\alpha}}$, for all $i \in I \setminus J$, and (iii) $\Theta = \sum_{j \in J} p_j \cdot \Theta_j$. Note that if $\alpha \neq \tau$ then this definition admits that only some terms in the support of Δ make the $\xrightarrow{\hat{\alpha}}$ transition. Then, we define the *weak transition relation* $\stackrel{\hat{\tau}}{\Longrightarrow}$ as the transitive and reflexive closure of $\xrightarrow{\hat{\tau}}$, *i.e.*, $\stackrel{\hat{\tau}}{\Longrightarrow} = (\xrightarrow{\hat{\tau}})^*$, while for $\alpha \neq \tau$ we let $\stackrel{\hat{\alpha}}{\Longrightarrow}$ denote $\stackrel{\hat{\tau}}{\Longrightarrow}\xrightarrow{\hat{\alpha}}\stackrel{\hat{\tau}}{\Longrightarrow}$.

2.2 Timed Weak Bisimulation with Tolerance

In this section, we define a family of relations $\approx_p^k$ over $\mathcal{T}$, with $p \in [0, 1]$ and $k \in \mathbb{N}^+ \cup \{\infty\}$, where, intuitively, $t \approx_p^k t'$ means that *t and t' can weakly bisimulate each other with a tolerance p accumulated in k timed steps.* This is done by introducing a family of *pseudometrics* $\mathbf{m}^k : \mathcal{T} \times \mathcal{T} \to [0, 1]$ and defining $t \approx_p^k t'$ iff $\mathbf{m}^k(t, t') = p$. The pseudometrics $\mathbf{m}^k$ will have the following properties for any $t, t' \in \mathcal{T}$: (i) $\mathbf{m}^{k_1}(t, t') \leq \mathbf{m}^{k_2}(t, t')$ whenever $k_1 < k_2$ (tolerance monotonicity); (ii) $\mathbf{m}^\infty(t, t') = p$ iff p is the distance between t and t' as given by the weak bisimilarity metric in [5] in an untimed setting; (iiii) $\mathbf{m}^\infty(t, t') = 0$ iff t and t' are related by the standard weak probabilistic bisimilarity [27].

Let us recall the standard definition of pseudometric.

Definition 2 (Pseudometric). *A function $d : \mathcal{T} \times \mathcal{T} \to [0, 1]$ is a* 1-bounded pseudometric *over $\mathcal{T}$ if*

- *$d(t, t) = 0$ for all $t \in \mathcal{T}$,*
- *$d(t, t') = d(t', t)$ for all $t, t' \in \mathcal{T}$ (symmetry),*
- *$d(t, t') \leq d(t, t'') + d(t'', t')$ for all $t, t', t'' \in \mathcal{T}$ (triangle inequality).*

In order to define the family of functions $\mathbf{m}^k$, we define an auxiliary family of functions $\mathbf{m}^{k,h} : \mathcal{T} \times \mathcal{T} \to [0, 1]$, with $k, h \in \mathbb{N}$, quantifying the tolerance of the weak bisimulation after a sequence of computation steps such that: (i) the sequence contains exactly k tick-actions, (ii) the sequence terminates with a tick-action, (iii) any term performs exactly h untimed actions before the first

tick-action, (iv) between any i-th and $(i+1)$-th tick-action, with $1 \leq i < k$, there are an arbitrary number of untimed actions.

The definition of $\mathbf{m}^{k,h}$ relies on a *timed and quantitative* version of the classic bisimulation game: The tolerance between t and t' as given by $\mathbf{m}^{k,h}(t,t')$ can be below a threshold $\epsilon \in [0,1]$ only if each transition $t \xrightarrow{\alpha} \Delta$ is mimicked by a weak transition $t' \xRightarrow{\hat{\alpha}} \Theta$ such that the bisimulation tolerance between Δ and Θ is, in turn, below ϵ. This requires to lift pseudometrics over $\mathcal{T}$ to pseudometrics over (sub-)distributions in $\mathcal{D}_{\mathrm{sub}}(\mathcal{T})$. To this end, we adopt the notions of *matching* [37] (also called coupling) and *Kantorovich lifting* [4].

Definition 3 (Matching). *A* matching *for a pair of distributions* $(\Delta, \Theta) \in \mathcal{D}(\mathcal{T}) \times \mathcal{D}(\mathcal{T})$ *is a distribution* ω *in the state product space* $\mathcal{D}(\mathcal{T} \times \mathcal{T})$ *such that:*

- $\sum_{t' \in \mathcal{T}} \omega(t,t') = \Delta(t)$, *for all* $t \in \mathcal{T}$, *and*
- $\sum_{t \in \mathcal{T}} \omega(t,t') = \Theta(t')$, *for all* $t' \in \mathcal{T}$.

We write $\Omega(\Delta, \Theta)$ *to denote the set of all matchings for* (Δ, Θ).

A matching for (Δ, Θ) may be understood as a transportation schedule for the shipment of probability mass from Δ to Θ [37].

Definition 4 (Kantorovich lifting). *Assume a pseudometric* $d \colon \mathcal{T} \times \mathcal{T} \to [0,1]$. *The* Kantorovich lifting *of* d *is the function* $\mathbf{K}(d) \colon \mathcal{D}(\mathcal{T}) \times \mathcal{D}(\mathcal{T}) \to [0,1]$ *defined for distributions* Δ *and* Θ *as:*

$$\mathbf{K}(d)(\Delta, \Theta) \stackrel{def}{=} \min_{\omega \in \Omega(\Delta,\Theta)} \sum_{s,t \in \mathcal{T}} \omega(s,t) \cdot d(s,t).$$

Note that since we are considering only distributions with finite support, the minimum over the set of matchings $\Omega(\Delta, \Theta)$ used in Definition 4 is well defined.

Pseudometrics $\mathbf{m}^{k,h}$ are inductively defined on k and h by means of suitable *functionals* over the complete lattice $([0,1]^{\mathcal{T} \times \mathcal{T}}, \sqsubseteq)$ of functions of type $\mathcal{T} \times \mathcal{T} \to [0,1]$, ordered by $d_1 \sqsubseteq d_2$ iff $d_1(t,t') \leq d_2(t,t')$ for all $t, t' \in \mathcal{T}$. Notice that in this lattice, for each set $D \subseteq [0,1]^{\mathcal{T} \times \mathcal{T}}$, the supremum and infimum are defined as $\sup(D)(t,t') = \sup_{d \in D} d(t,t')$ and $\inf(D)(t,t') = \inf_{d \in D} d(t,t')$, for all $t, t' \in \mathcal{T}$. The infimum of the lattice is the constant function zero, denoted by $\mathbf{0}$, and the supremum is the constant function one, denoted by $\mathbf{1}$.

Definition 5 (Functionals for $\mathbf{m}^{k,h}$). *The functionals* $\mathbf{B}, \mathbf{B}_{\mathsf{tick}} \colon [0,1]^{\mathcal{T} \times \mathcal{T}} \to [0,1]^{\mathcal{T} \times \mathcal{T}}$ *are defined for any function* $d \in [0,1]^{\mathcal{T} \times \mathcal{T}}$ *and terms* $t, t' \in \mathcal{T}$ *as:*

$$\begin{aligned}
\mathbf{B}(d)(t,t') = \max\{\, & d(t,t'), \\
& \sup_{\alpha \in \mathbf{A} \setminus \{\mathsf{tick}\}} \ \max_{t \xrightarrow{\alpha} \Delta} \ \inf_{t' \xRightarrow{\hat{\alpha}} \Theta} \mathbf{K}(d)\big(\Delta, \Theta + (1-|\Theta|)\overline{\mathsf{Dead}}\big), \\
& \sup_{\alpha \in \mathbf{A} \setminus \{\mathsf{tick}\}} \ \max_{t' \xrightarrow{\alpha} \Theta} \ \inf_{t \xRightarrow{\hat{\alpha}} \Delta} \mathbf{K}(d)\big(\Delta + (1-|\Delta|)\overline{\mathsf{Dead}}, \Theta\big) \,\} \\
\mathbf{B}_{\mathsf{tick}}(d)(t,t') = \max\{\, & d(t,t'), \\
& \max_{t \xrightarrow{\mathsf{tick}} \Delta} \ \inf_{t' \xRightarrow{\widehat{\mathsf{tick}}} \Theta} \mathbf{K}(d)\big(\Delta, \Theta + (1-|\Theta|)\overline{\mathsf{Dead}}\big), \\
& \max_{t' \xrightarrow{\mathsf{tick}} \Theta} \ \inf_{t \xRightarrow{\widehat{\mathsf{tick}}} \Delta} \mathbf{K}(d)\big(\Delta + (1-|\Delta|)\overline{\mathsf{Dead}}, \Theta\big) \,\}
\end{aligned}$$

where $\inf \emptyset = 1$ *and* $\max \emptyset = 0$.

Notice that all max in Definition 5 are well defined since the pLTS is image-finite. Notice also that any strong transitions from t to a distribution Δ is mimicked by a weak transition from t', which, in general, takes to a sub-distribution Θ. Thus, process t' may not simulate t with probability $1-|\Theta|$.

Definition 6 (Timed weak bisimilarity metrics). *The family of the* timed weak bisimilarity metrics $\mathbf{m}^k\colon (\mathcal{T}\times\mathcal{T})\to[0,1]$ *is defined for all* $k\in\mathbb{N}$ *by*
$\mathbf{m}^k = \begin{cases}\mathbf{0} & \text{if } k=0\\ \sup_{h\in\mathbb{N}}\mathbf{m}^{k,h} & \text{if } k>0\end{cases}$ *while the functions* $\mathbf{m}^{k,h}\colon(\mathcal{T}\times\mathcal{T})\to[0,1]$ *are defined for all* $k\in\mathbb{N}^+$ *and* $h\in\mathbb{N}$ *by*
$\mathbf{m}^{k,h} = \begin{cases}\mathbf{B}_{\mathsf{tick}}(\mathbf{m}^{k-1}) & \text{if } h=0\\ \mathbf{B}(\mathbf{m}^{k,h-1}) & \text{if } h>0.\end{cases}$ *Then, we define* $\mathbf{m}^\infty\colon(\mathcal{T}\times\mathcal{T})\to[0,1]$ *as* $\mathbf{m}^\infty=\sup_{k\in\mathbb{N}}\mathbf{m}^k$.

Note that any $\mathbf{m}^{k,h}$ is obtained from $\mathbf{m}^{k-1}$ by one application of the functional $\mathbf{B}_{\mathsf{tick}}$, in order to take into account the distance between terms introduced by the k-th tick-action, and h applications of the functional $\mathbf{B}$, in order to lift such a distance to terms that take h untimed actions to be able to perform a tick-action. By taking $\sup_{h\in\mathbb{N}}\mathbf{m}^{k,h}$ we consider an arbitrary number of untimed steps.

The pseudometric property of $\mathbf{m}^k$ is necessary to conclude that the tolerance between terms as given by $\mathbf{m}^k$ is a reasonable notion of behavioural distance.

Theorem 1. *For any* $k\geq 1$, $\mathbf{m}^k$ *is a 1-bounded pseudometric.*

Finally, everything is in place to define our timed weak bisimilarity $\approx_p^k$ with tolerance $p\in[0,1]$ accumulated after k time units, for $k\in\mathbb{N}\cup\{\infty\}$.

Definition 7 (Timed weak bisimilarity with tolerance). *Let* $t,t'\in\mathcal{T}$, $k\in\mathbb{N}$ *and* $p\in[0,1]$. *We say that* t and t' are weakly bisimilar with a tolerance p, which accumulates in k timed actions, *written* $t\approx_p^k t'$, *if and only if* $\mathbf{m}^k(t,t')=p$. *Then, we write* $t\approx_p^\infty t'$ *if and only if* $\mathbf{m}^\infty(t,t')=p$.

Since the Kantorovich lifting $\mathbf{K}$ is monotone [26], it follows that both functionals $\mathbf{B}$ and $\mathbf{B}_{\mathsf{tick}}$ are monotone. This implies that, for any $k\geq 1$, $(\mathbf{m}^{k,h})_{h\geq 0}$ is a non-decreasing chain and, analogously, also $(\mathbf{m}^k)_{k\geq 0}$ is a non-decreasing chain, thus giving the following expected result saying that the distance between terms grows when we consider a higher number of tick computation steps.

Proposition 1 (Tolerance monotonicity). *For all terms* $t,t'\in\mathcal{T}$ *and* $k_1,k_2\in\mathbb{N}^+$ *with* $k_1<k_2$, $t\approx_{p_1}^{k_1}t'$ *and* $t\approx_{p_2}^{k_2}t'$ *entail* $p_1\leq p_2$.

We conclude this section by comparing our behavioural distance with the behavioural relations known in the literature.

We recall that in [5] a family of relations $\simeq_p$ for *untimed* process calculi are defined such that $t\simeq_p t'$ if and only if t and t' weakly bisimulate each other with tolerance p. Of course, one can apply these relations also to timed process calculi, the effect being that timed actions are treated in exactly the

same manner as untimed actions. The following result compares the behavioural metrics proposed in the present paper with those of [5], and with the classical notions of probabilistic weak bisimilarity [27] denoted $\approx$.

Proposition 2. *Let $t, t' \in \mathcal{T}$ and $p \in [0,1]$. Then,*

- $t \approx_p^\infty t'$ *iff* $t \simeq_p t'$
- $t \approx_0^\infty t'$ *iff* $t \approx t'$.

3 A Simple Probabilistic Timed Calculus for IoT Systems

In this section, we propose a simple extension of Hennessy and Regan's *timed process algebra* TPL [12] to express *IoT systems* and *cyber-physical attacks*. The goal is to show that timed weak bisimilarity with tolerance is a suitable notion to estimate the impact of cyber-physical attacks on IoT systems.

Let us start with some preliminary notations.

Notation 1. *We use x, x_k for* state variables*, c, c_k, for* communication channels*, z, z_k for* communication variables*, s, s_k for* sensors devices*, while o ranges over both channels and sensors.* Values*, ranged over by v, v', belong to a* finite *set of admissible values $\mathcal{V}$. We use u, u_k for both values and communication variables. Given a generic set of names $\mathcal{N}$, we write $\mathcal{V}^{\mathcal{N}}$ to denote the set of functions $\mathcal{N} \to \mathcal{V}$ assigning a value to each name in $\mathcal{N}$. For $m \in \mathbb{N}$ and $n \in \mathbb{N} \cup \{\infty\}$, we write $m..n$ to denote an* integer interval*. As we will adopt a discrete notion of time, we will use integer intervals to denote* time intervals*.*

State variables are associated to physical properties like *temperature*, *pressure*, etc. *Sensor names* are metavariables for sensor devices, such as *thermometers* and *barometers*. Please, notice that in cyber-physical systems, state variables cannot be directly accessed but they can only be tested via one or more sensors.

Definition 8 (IoT system). *Let $\mathcal{X}$ be a set of state variables and $\mathcal{S}$ be a set of sensors. Let $range : \mathcal{X} \to 2^{\mathcal{V}}$ be a total function returning the range of admissible values for any state variable $x \in \mathcal{X}$. An* IoT system *consists of two components:*

- *a* physical environment $\xi = \langle \xi_{\rm x}, \xi_{\rm m} \rangle$ *where:*
  - $\xi_{\rm x} \in \mathcal{V}^{\mathcal{X}}$ *is the* physical state *of the system that associates a value to each state variable in $\mathcal{X}$, such that $\xi_{\rm x}(x) \in range(x)$ for any $x \in \mathcal{X}$,*
  - $\xi_{\rm m} : \mathcal{V}^{\mathcal{X}} \to \mathcal{S} \to \mathcal{D}(\mathcal{V})$ *is the* measurement map *that given a physical state returns a function that associates to any sensor in $\mathcal{S}$ a discrete probability distribution over the set of possible sensed values;*
- *a* logical (or cyber) component P *that interacts with the sensors defined in ξ, and can communicate, via channels, with other cyber components.*

We write $\xi \bowtie P$ to denote the resulting IoT system, and use M and N to range over IoT systems.

Let us now formalise the *cyber component* of an IoT system. Basically, we adapt Hennessy and Regan's *timed process algebra TPL* [12].

Definition 9 (Logics). Logical components *of IoT systems are defined by the following grammar:*

$$P, Q \ ::= \ \mathsf{nil} \ \big| \ \mathsf{tick}.P \ \big| \ P \parallel Q \ \big| \ \lfloor \mathit{pfx}.P \rfloor Q \ \big| \ H\langle \tilde{u} \rangle \ \big| \ \mathsf{if}\,(b)\,\{P\}\,\mathsf{else}\,\{Q\} \ \big| \ P\backslash c$$
$$\mathit{pfx} \ ::= \ o!v \ \big| \ o?(z)$$

The process $\mathsf{tick}.P$ sleeps for one time unit and then continues as P. We write $P \parallel Q$ to denote the *parallel composition* of concurrent processes P and Q. The process $\lfloor \mathit{pfx}.P \rfloor Q$ denotes *prefixing with timeout*. We recall that o ranges over both channel and sensor names. Thus, for instance, $\lfloor c!v.P \rfloor Q$ sends the value v on channel c and, after that, it continues as P; otherwise, if no communication partner is available within one time unit, it evolves into Q. The process $\lfloor c?(z).P \rfloor Q$ is the obvious counterpart for channel reception. On the other hand, the process $\lfloor s?(z).P \rfloor Q$ reads the sensor s, according to the measurement map of the systems, and, after that, it continues as P. The process $\lfloor s!v.P \rfloor Q$ writes to the sensor s and, after that, it continues as P; here, we wish to point out that this a *malicious activity*, as controllers may only access sensors for reading sensed data. Thus, the construct $\lfloor s!v.P \rfloor Q$ serves to implement an *integrity attack* that attempts at synchronising with the controller of sensor s to provide a fake value v. In the following, we say that a process is *honest* if it never writes on sensors. The definition of honesty naturally lifts to IoT systems. In processes of the form $\mathsf{tick}.Q$ and $\lfloor \mathit{pfx}.P \rfloor Q$, the occurrence of Q is said to be *time-guarded*. *Recursive processes* $H\langle \tilde{u} \rangle$ are defined via equations $H(z_1, \ldots, z_k) = P$, where (i) the tuple $z_1, \ldots, z_k$ contains all the variables that appear free in P, and (ii) P contains *only time-guarded occurrences* of the process identifiers, such as H itself (to avoid *zeno behaviours*). The two remaining constructs are standard; they model conditionals and channel restriction, respectively.

Finally, we define how to compose IoT systems. For simplicity, we compose two systems only if they have the same physical environment.

Definition 10 (System composition). *Let $M_1 = \xi \bowtie P_1$ and $M_2 = \xi \bowtie P_2$ be two IoT systems, and Q be a process whose sensors are defined in the physical environment ξ. We write:*

- *$M_1 \parallel M_2$ to denote $\xi \bowtie (P_1 \parallel P_2)$;*
- *$M_1 \parallel Q$ to denote $\xi \bowtie (P_1 \parallel Q)$;*
- *$M_1\backslash c$ as an abbreviation for $\xi \bowtie (P_1\backslash c)$.*

We conclude this section with the following abbreviations that will be used in the rest of the paper.

Notation 2. *We write $P\backslash\{c_1, c_2, \ldots, c_n\}$, or $P\backslash\tilde{c}$, to mean $P\backslash c_1\backslash c_2 \cdots \backslash c_n$. For simplicity, we sometimes abbreviate both $H(i)$ and $H\langle i \rangle$ with H_i. We write $\mathit{pfx}.P$ as an abbreviation for the process defined via the equation $H = \lfloor \mathit{pfx}.P \rfloor H$, where the process name H does not occur in P. We write $\mathsf{tick}^k.P$ as a shorthand for $\mathsf{tick}.\mathsf{tick}.\ldots.\mathsf{tick}.P$, where the prefix tick appears $k \geq 0$ consecutive times. We write Dead to denote a deadlocked IoT system that cannot perform any action.*

Table 1. Labelled transition system2 for processes

$$\text{(Write)}\ \frac{-}{\lfloor o!v.P \rfloor Q \xrightarrow{o!v} P} \qquad \text{(Read)}\ \frac{-}{\lfloor o?(z).P \rfloor Q \xrightarrow{o?(z)} P}$$

$$\text{(Sync)}\ \frac{P \xrightarrow{o!v} P' \quad Q \xrightarrow{o?(z)} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'\{^v/_z\}} \qquad \text{(Par)}\ \frac{P \xrightarrow{\lambda} P' \quad \lambda \neq \mathsf{tick}}{P \parallel Q \xrightarrow{\lambda} P' \parallel Q}$$

$$\text{(Res)}\ \frac{P \xrightarrow{\lambda} P' \quad \lambda \notin \{o!v, o?(z)\}}{P \backslash o \xrightarrow{\lambda} P' \backslash o} \qquad \text{(Rec)}\ \frac{P\{^{\tilde{v}}/_{\tilde{z}}\} \xrightarrow{\lambda} Q \quad H(\tilde{z}) = P}{H\langle \tilde{v} \rangle \xrightarrow{\lambda} Q}$$

$$\text{(Then)}\ \frac{[\![b]\!] = \mathsf{true} \quad P \xrightarrow{\lambda} P'}{\mathsf{if}\ (b)\ \{P\}\ \mathsf{else}\ \{Q\} \xrightarrow{\lambda} P'} \qquad \text{(Else)}\ \frac{[\![b]\!] = \mathsf{false} \quad Q \xrightarrow{\lambda} Q'}{\mathsf{if}\ (b)\ \{P\}\ \mathsf{else}\ \{Q\} \xrightarrow{\lambda} Q'}$$

$$\text{(TimeNil)}\ \frac{-}{\mathsf{nil} \xrightarrow{\mathsf{tick}} \mathsf{nil}} \qquad \text{(Delay)}\ \frac{-}{\mathsf{tick}.P \xrightarrow{\mathsf{tick}} P}$$

$$\text{(Timeout)}\ \frac{-}{\lfloor pfx.P \rfloor Q \xrightarrow{\mathsf{tick}} Q} \qquad \text{(TimePar)}\ \frac{P \xrightarrow{\mathsf{tick}} P' \quad Q \xrightarrow{\mathsf{tick}} Q'}{P \parallel Q \xrightarrow{\mathsf{tick}} P' \parallel Q'}$$

3.1 Probabilistic Labelled Transition Semantics

As said before, sensors serve to observe the evolution of the physical state of an IoT system. However, sensors are usually affected by an *error/noise* that we represent in our measurement maps by means of discrete probability distributions. For this reason, we equip our calculus with a probabilistic labelled transition system. In the following, the symbol ϵ ranges over distributions on physical environments, whereas π ranges over distributions on (logical) processes. Thus, $\epsilon \bowtie \pi$ denotes the distribution over IoT systems defined by $(\epsilon \bowtie \pi)(\xi \bowtie P) = \epsilon(\xi)\cdot\pi(P)$. The symbol γ ranges over distributions on IoT systems.

In Table 1, we give a standard labelled transition system for logical components (timed processes), whereas in Table 2 we rely on the LTS of Table 1 to define a simple pLTS for IoT systems by lifting transition rules from processes to systems.

In Table 1, the meta-variable λ ranges over labels in the set $\{\tau, \mathsf{tick}, o!v, o?(z)\}$. Rule (Sync) serve to model synchronisation and value passing, on some name (for channel or sensor) o: if o is a channel then we have standard point-to-point communication, whereas if o is a sensor then this rule models an *integrity attack* on sensor s, as the controller is provided with a fake value v. The remaining rules are standard. The symmetric counterparts of rules (Sync) and (Par) are omitted.

According to Table 2, IoT systems may fire four possible actions ranged over by α. These actions represent: internal activities (τ), the passage of time (tick), channel transmission ($c!v$) and channel reception ($c?v$).

Rules (Snd) and (Rcv) model transmission and reception on a channel c with an external system, respectively. Rule (SensRead) models the reading of

Table 2. Probabilistic LTS for a IoT system $\xi \bowtie P$ with $\xi = \langle \xi_x, \xi_m \rangle$

$$\text{(Snd)}\ \frac{P \xrightarrow{c!v} P'}{\xi \bowtie P \xrightarrow{c!v} \overline{\xi} \bowtie \overline{P'}} \qquad \text{(Rcv)}\ \frac{P \xrightarrow{c?(z)} P'}{\xi \bowtie P \xrightarrow{c?v} \overline{\xi} \bowtie \overline{P'\{^v/_z\}}}$$

$$\text{(SensRead)}\ \frac{P \xrightarrow{s?(z)} P' \quad \xi_m(\xi_x)(s) = \sum_{i \in I} p_i \cdot \overline{v_i}}{\xi \bowtie P \xrightarrow{\tau} \overline{\xi} \bowtie \sum_{i \in I} p_i \cdot \overline{P'\{^{v_i}/_z\}}}$$

$$\text{(Tau)}\ \frac{P \xrightarrow{\tau} P'}{\xi \bowtie P \xrightarrow{\tau} \overline{\xi} \bowtie \overline{P'}} \qquad \text{(Time)}\ \frac{P \xrightarrow{\mathsf{tick}} P' \quad \xi \bowtie P \not\xrightarrow{\tau} \quad \xi' \in next(\xi)}{\xi \bowtie P \xrightarrow{\mathsf{tick}} \overline{\xi'} \bowtie \overline{P'}}$$

the value detected at a *sensor* s according to the current physical environment $\xi = \langle \xi_x, \xi_m \rangle$. In particular, this rule says that if a process P in a system $\xi \bowtie P$ reads a sensor s defined in ξ then it will get a value that may vary according to the probability distribution resulting by providing the state function ξ_x and the sensor s to the measurement map ξ_m.

Rule (Tau) lifts internal actions from processes to systems. This includes communications on channels and malicious accesses to sensors' controllers. According to Definition 10, rule (Tau) models also channel communication between two parallel IoT systems sharing the same physical environment.

A second lifting occurs in rule (Time) for timed actions tick. Here, ξ' denotes an admissible physical environment for the next time slot, nondeterministically chosen from the *finite* set $next(\langle \xi_x, \xi_m \rangle)$. This set is defined as $\{\langle \xi'_x, \xi_m \rangle : \xi'_x(x) \in range(x) \text{ for any } x \in \mathcal{X}\}$.[1] As a consequence, the rules in Table 2 define an *image-finite* pLTS.

For simplicity, we abstract from the *physical process* behind our IoT systems.

4 Cyber-Physical Attacks on Sensor Devices

In this section, we consider attacks tampering with sensors by eavesdropping and possibly modifying the sensor measurements provided to the corresponding controllers. These attacks may affect both the *integrity* and the *availability* of the system under attack. We do not represent (well-known) attacks on communication channels as our focus is on attacks to physical devices and the consequent impact on the physical state. However, our technique can be easily generalised to deal with attacks on channels as well.

Definition 11 (Cyber-physical attack). *A (pure) cyber-physical attack A is a process derivable from the grammar of Definition 9 such that:*

- *A writes on at least one sensor;*
- *A never uses communication channels.*

[1] The finiteness follows from the finiteness of $\mathcal{V}$, and hence of $range(x)$, for any $x \in \mathcal{X}$.

In order to make security assessments on our IoT systems, we adapt a well-known approach called *Generalized Non Deducibility on Composition (GNDC)* [7]. Intuitively, an attack A affects an honest IoT system M if the execution of the composed system $M \parallel A$ differs from that of the original system M in an observable manner. Basically, a cyber-physical attack can influence the system under attack in at least two different ways:

– The system $M \parallel A$ might have non-genuine execution traces containing observables that cannot be reproduced by M; here the attack affects the *integrity* of the system behaviour (*integrity attack*).
– The system M might have execution traces containing observables that cannot be reproduced by the system under attack $M \parallel A$ (because they are prevented by the attack); this is an attack against the *availability* of the system (*DoS attack*).

Now, everything is in place to provide a formal definition of *system tolerance* and *system vulnerability* with respect to a given attack. Intuitively, a system M tolerates an attack A if the presence of the attack does not affect the behaviour of M; on the other hand M is vulnerable to A in a certain time interval if the attack has an *impact* on the behaviour of M in that time interval.

Definition 12 (Attack tolerance). *Let M be a honest IoT system. We say that M* tolerates an attack A *if* $M \parallel A \approx_0^\infty M$.

Definition 13 (Attack vulnerability and impact). *Let M be a honest IoT system. We say that M is* vulnerable to an attack A in the time interval $m..n$ with *impact* $p \in [0,1]$, *for* $m \in \mathbb{N}^+$ *and* $n \in \mathbb{N}^+ \cup \{\infty\}$, *if* $m..n$ *is the smallest time interval such that: (i)* $M \parallel A \approx_0^{m-1} M$, *(ii)* $M \parallel A \approx_p^n M$, *(iii)* $M \parallel A \approx_p^\infty M$.[2]

Basically, the definition above says that if a system is vulnerable to an attack in the time interval $m..n$ then the perturbation introduced by the attack starts in the m-th time slot and reaches the maximum impact in the n-th time slot.

The following result says that both notions of tolerance and vulnerability are suitable for *compositional reasonings*. More precisely, we prove that they are both preserved by parallel composition and channel restriction. Actually, channel restriction may obviously make a system less vulnerable by hiding channels.

Theorem 2 (Compositionality). *Let $M_1 = \xi \bowtie P_1$ and $M_2 = \xi \bowtie P_2$ be two honest IoT systems with the same physical environment ξ, A an arbitrary attack, and $\tilde{c}$ a set of channels.*

– *If both M_1 and M_2 tolerate A then $(M_1 \parallel M_2)\backslash\tilde{c}$ tolerates A.*
– *If M_1 is vulnerable to A in the time interval $m_1..n_1$ with impact p_1, and M_2 is vulnerable to A in the time interval $m_2..n_2$ with impact p_2, then $M_1 \parallel M_2$ is vulnerable to A in a the time interval* $\min(m_1, m_2)..\max(n_1, n_2)$ *with an impact* $p' \leq (p_1 + p_2 - p_1 p_2)$.

[2] By Proposition 1, at all time instants greater than n the impact remains p.

– *If M_1 is vulnerable to A in the interval $m_1..n_1$ with impact p_1 then $M_1 \backslash \tilde{c}$ is vulnerable to A in a time interval $m'..n' \subseteq m_1..n_1$ with an impact $p' \le p_1$.*

Note that if an attack A is tolerated by a system M and can interact with a honest process P then the compound system $M \parallel P$ may be vulnerable to A. However, if A does not write on the sensors of P then it is tolerated by $M \parallel P$ as well. The bound $p' \le (p_1 + p_2 - p_1 p_2)$ can be explained as follows. The likelihood that the attack does not impact on M_i is $(1 - p_i)$, for $i \in \{1, 2\}$. Thus, the likelihood that the attack impacts neither on M_1 nor on M_2 is at least $(1 - p_1)(1 - p_2)$. Summarising, the likelihood that the attack impacts on at least one of the two systems M_1 and M_2 is at most $1 - (1 - p_1)(1 - p_2) = p_1 + p_2 - p_1 p_2$.

An easy corollary of Theorem 2 allows us to lift the notions of tolerance and vulnerability from a honest system M to the compound systems $M \parallel P$, for a honest process P.

Corollary 1. *Let M be a honest system, A an attack, $\tilde{c}$ a set of channels, and P a honest process that reads sensors defined in M but not those written by A.*

– *If M tolerates A then $(M \parallel P) \backslash \tilde{c}$ tolerates A.*
– *If M is vulnerable to A in the interval $m..n$ with impact p, then $(M \parallel P) \backslash \tilde{c}$ is vulnerable to A in a time interval $m'..n' \subseteq m..n$, with an impact $p' \le p$.*

5 Attacking a Smart Surveillance System: A Case Study

Consider an alarmed ambient consisting of three rooms, r_i for $i \in \{1, 2, 3\}$, each of which equipped with a sensor s_i to detect unauthorised accesses. The alarm goes off if at least one of the three sensors detects an intrusion.

The logics of the system can be easily specified in our language as follows:

$$
\begin{aligned}
\mathit{Sys} &= (\mathit{Mng} \parallel \mathit{Ctrl_1} \parallel \mathit{Ctrl_2} \parallel \mathit{Ctrl_3}) \backslash \{c_1, c_2, c_3\} \\
\mathit{Mng} &= c_1?(z_1).c_2?(z_2).c_3?(z_3).\mathsf{if}\ (\textstyle\bigvee_{i=1}^{3} z_i{=}\mathsf{on})\ \{\mathit{alarm}!\mathsf{on}.\mathsf{tick}.\mathit{Check_k}\}\ \mathsf{else}\ \{\mathsf{tick}.\mathit{Mng}\} \\
\mathit{Check_0} &= \mathit{Mng} \\
\mathit{Check_j} &= \mathit{alarm}!\mathsf{on}.c_1?(z_1).c_2?(z_2).c_3?(z_3).\mathsf{if}\ (\textstyle\bigvee_{i=1}^{3} z_i = \mathsf{on})\ \{\mathsf{tick}.\mathit{Check_k}\} \\
&\quad\ \mathsf{else}\ \{\mathsf{tick}.\mathit{Check_{j-1}}\} \quad \text{for } j > 0 \\
\mathit{Ctrl_i} &= s_i?(z_i).\mathsf{if}\ (z_i{=}\mathsf{presence})\ \{c_i!\mathsf{on}.\mathsf{tick}.\mathit{Ctrl_i}\}\ \mathsf{else}\ \{c_i!\mathsf{off}.\mathsf{tick}.\mathit{Ctrl_i}\}\ \text{for } i{\in}\{1, 2, 3\}.
\end{aligned}
$$

Intuitively, the process Sys is composed by three controllers, $\mathit{Ctrl_i}$, one for each sensor s_i, and a manager Mng that interacts with the controllers via private channels c_i. The process Mng fires an alarm if at least one of the controllers signals an intrusion. As usual in this kind of surveillance systems, the alarm will keep going off for k instants of time after the last detected intrusion.

As regards the physical environment, the physical state $\xi_{\mathrm{x}} : \{r_1, r_2, r_3\} \rightarrow \{\mathsf{presence}, \mathsf{absence}\}$ is set to $\xi_{\mathrm{x}}(r_i) = \mathsf{absence}$, for any $i \in \{1, 2, 3\}$. Furthermore, let p_i^+ and p_i^- be the probabilities of having *false positives* (erroneously detected intrusion) and *false negatives* (erroneously missed intrusion) at sensor s_i[3], respectively, for $i \in \{1, 2, 3\}$, the measurement function ξ_{m} is defined

[3] These probabilities are usually very small; we assume them smaller than $\frac{1}{2}$.

as follows: $\xi_{\rm m}(\xi_{\rm x})(s_i) = (1-p_i^-)\,\overline{\mathsf{presence}} + p_i^-\overline{\mathsf{absence}}$, if $\xi_{\rm x}(r_i) = \mathsf{presence}$; $\xi_{\rm m}(\xi_{\rm x})(s_i) = (1-p_i^+)\,\overline{\mathsf{absence}} + p_i^+\overline{\mathsf{presence}}$, otherwise.

Thus, the whole IoT system has the form $\xi \bowtie \mathit{Sys}$, with $\xi = \langle \xi_{\rm x}, \xi_{\rm m} \rangle$.

We start our analysis studying the impact of a simple cyber-physical attack that provides fake *false positives* to the controller of one of the sensors s_i. This attack affects the *integrity* of the system behaviour as the system under attack will fire alarms without any physical intrusion.

Example 1 (Introducing false positives). In this example, we provide an attack that tries to increase the number of false positives detected by the controller of some sensor s_i during a specific time interval $m..n$, with $m, n \in \mathbb{N}$, $n \geq m > 0$. Intuitively, the attack waits for $m-1$ time slots, then, during the time interval $m..n$, it provides the controller of sensor s_i with a fake intrusion signal. Formally,

$$\begin{aligned} A_{\mathsf{fp}}(i,m,n) &= \mathsf{tick}^{m-1}.B\langle i, n-m+1\rangle \\ B(i,j) &= \mathsf{if}\ (j=0)\ \{\mathsf{nil}\}\ \mathsf{else}\ \{\lfloor s_i!\mathsf{presence}.\mathsf{tick}.B\langle i,j-1\rangle\rfloor B\langle i,j-1\rangle\}. \end{aligned}$$

In the following proposition, we use our metric to measure the perturbation introduced by the attack to the controller of a sensor s_i by varying the time of observation of the system under attack.

Proposition 3. *Let ξ be an arbitrary physical state for the systems $M_i = \xi \bowtie \mathit{Ctrl}_i$, for $i \in \{1,2,3\}$. Then,*

- $M_i \parallel A_{\mathsf{fp}}\langle i,m,n\rangle \approx_0^j M_i$, *for $j \in 1..m{-}1$;*
- $M_i \parallel A_{\mathsf{fp}}\langle i,m,n\rangle \approx_h^j M_i$, *with $h = 1-(p_i^+)^{j-m+1}$, for $j \in m..n$;*
- $M_i \parallel A_{\mathsf{fp}}\langle i,m,n\rangle \approx_r^j M_i$, *with $r = 1-(p_i^+)^{n-m+1}$, for $j > n$ or $j = \infty$.*

By an application of Definition 13 we can measure the impact of the attack A_{fp} to the (sub)systems $\xi \bowtie \mathit{Ctrl}_i$.

Corollary 2. *The IoT systems $\xi \bowtie \mathit{Ctrl}_i$ are vulnerable to the attack $A_{\mathsf{fp}}\langle i,m,n\rangle$ in the time interval $m..n$ with impact $1-(p_i^+)^{n-m+1}$.*

Note that the vulnerability window $m..n$ coincides with the activity period of the attack A_{fp}. This means that the system under attack recovers its normal behaviour immediately after the termination of the attack. However, in general, an attack may impact the behaviour of the target system long after its termination.

Note also that the attack $A_{\mathsf{fp}}\langle i,m,n\rangle$ has an impact not only on the controller Ctrl_i but also on the whole system $\xi \bowtie \mathit{Sys}$. This because the process Mng will surely fire the alarm as it will receive at least one intrusion detection from Ctrl_i. However, by an application of Corollary 1 we can prove that the impact on the whole system will not get amplified.

Proposition 4 (Impact of the attack A_{fp}). *The system $\xi \bowtie \mathit{Sys}$ is vulnerable to the attack $A_{\mathsf{fp}}\langle i,m,n\rangle$ in a time interval $m'..n' \subseteq m..n$ with impact $p' \leq 1-(p_i^+)^{n-m+1}$.*

Now, the reader may wonder what happens if we consider a complementary attack that provides fake *false negatives* to the controller of one of the sensors s_i. In this case, the attack affects the *availability* of the system behaviour as the system will no fire the alarm in the presence of a real intrusion. This because a real intrusion will be somehow "hidden" by the attack.

Example 2 (Introducing false negatives). The goal of the following attack is to increase the number of false negatives during the time interval $m..n$, with $n \geq m > 0$. Formally, the attack is defined as follows:

$$A_{\mathsf{fn}}(i, m, n) = \mathsf{tick}^{m-1}.C\langle i, n-m+1\rangle$$
$$C(i, j) = \mathsf{if}\ (j = 0)\ \{\mathsf{nil}\}\ \mathsf{else}\ \{\lfloor s_i!\mathsf{absence}.\mathsf{tick}.C\langle i, j-1\rangle \rfloor C\langle i, j-1\rangle\}.$$

In the following proposition, we use our metric to measure the deviation introduced by the attack A_{fn} to the controller of a sensor s_i. With no surprise we get a result that is the symmetric version of Proposition 3.

Proposition 5. *Let ξ be an arbitrary physical state for the system $M_i = \xi \bowtie Ctrl_i$, for $i \in \{1, 2, 3\}$. Then,*

- $M_i \parallel A_{\mathsf{fn}}\langle i, m, n\rangle \approx_0^j M_i$, *for* $j \in 1..m-1$;
- $M_i \parallel A_{\mathsf{fn}}\langle i, m, n\rangle \approx_h^j M_i$, *with* $h = 1 - (p_i^-)^{j-m+1}$, *for* $j \in m..n$;
- $M_i \parallel A_{\mathsf{fn}}\langle i, m, n\rangle \approx_r^j M_i$, *with* $r = 1 - (p_i^-)^{n-m+1}$, *for* $j > n$ *or* $j = \infty$.

Again, by an application of Definition 13 we can measure the impact of the attack A_{fn} to the (sub)systems $\xi \bowtie Ctrl_i$.

Corollary 3. *The IoT systems $\xi \bowtie Ctrl_i$ are vulnerable to the attack $A_{\mathsf{fn}}\langle i, m, n\rangle$ in the time interval $m..n$ with impact $1 - (p_i^-)^{n-m+1}$.*

As our timed metric is compositional, by an application of Corollary 1 we can estimate the impact of the attack A_{fn} to the whole system $\xi \bowtie Sys$.

Proposition 6 (Impact of the attack A_{fn}). *The system $\xi \bowtie Sys$ is vulnerable to the attack $A_{\mathsf{fn}}\langle i, m, n\rangle$ in a time interval $m'..n' \subseteq m..n$ with impact $p' \leq 1 - (p_i^-)^{n-m+1}$.*

6 Conclusions, Related and Future Work

We have proposed a timed generalisation of the n-bisimulation metric [3], called *timed bisimulation metric*, obtained by defining two functionals over the complete lattice of the functions assigning a distance in [0,1] to each pair of systems: the former deals with the distance accumulated when executing untimed steps, the latter with the distance introduced by timed actions.

We have used our timed bisimulation metrics to provide a formal and *compositional* notion of *impact metric* for *cyber-physical attacks* on *IoT systems* specified in a simple timed process calculus. In particular, we have focussed on cyber-physical attacks targeting sensor devices (attack on sensors are by far the

most studied cyber-physical attacks [38]). We have used our timed weak bisimulation with tolerance to formalise the notions of *attack tolerance* and *attack vulnerability with a given impact* p. In particular, a system M is said to be vulnerable to an attack A in the time interval $m..n$ with impact p if the perturbation introduced by A becomes observable in the m-th time slot and yields the maximum impact p in the n-th time slot. Here, we wish to stress that the *vulnerability window* $m..n$ is quite informative. In practise, this interval says when an attack will produce observable effects on the system under attack. Thus, if n is finite we have an attack with *temporary effects*, otherwise we have an attack with *permanent effects*. Furthermore, if the attack is quick enough, and terminates well before the time instant m, then we have a *stealthy attack* that affects the system late enough to allow *attack camouflages* [11]. On the other hand, if at time m the attack is far from termination, then the IoT system under attack has good chances of undertaking countermeasures to stop the attack.

As a case study, we have estimated the impact of two cyber-physical attacks on sensors that introduce *false positives* and *false negatives*, respectively, into a simple surveillance system, affecting the *integrity* and the *availability* of the IoT system. Although our attacks are quite simple, the specification language and the corresponding metric semantics presented in the paper allow us to deal with smarter attacks, such as *periodic attacks* with constant or variable period of attack. Moreover, we can easily extend our threat model to recover (well-known) attacks on communication channels.

Related Work. A number of papers have recently proposed different methodologies for assessing the direct and indirect impact of attacks on CPSs.

Bilis et al. [1] proposed a systematic approach that uses five metrics derived from complex network theory to assess the impacts of cyber attacks on electric power systems. The metrics were used to rank nodes in a graph-based representation of an electric grid. Sgouras et al. [31] evaluated the impact of cyber attacks on a simulated smart metering infrastructure; the denial-of-service attacks against smart meters and utility servers caused severe communications interruptions. Sridhar and Govindarasu [33] evaluated the impacts of attacks on wide-area frequency control applications in power systems; their research showed that cyber attacks can significantly impact system stability by causing severe drops in system frequency. Genge et al. [10] introduced a methodology, inspired by research in system dynamics and sensitivity analysis, to compute the covariances of the observed variables before and after the execution of a specific intervention involving the control variables. Metrics are proposed for quantifying the significance of control variables and measuring the impact propagation of cyber attacks. Orojloo and Azgomi [25] investigated how an attack against system parameters can affect the values of other parameters. The system parameters are divided into two classes of cause and effect parameters, which can be same as or different from each other. They proposed metrics to prioritise the sensor readings and control signals based on their sensitivity to conducted attacks. Urbina et al. [35] defined an evaluation metric for attack-detection algo-

rithms that quantifies the negative impact of stealthy attacks and the inherent trade-off with false alarms. The authors showed that the impact of such attacks can be mitigated in several cases by the proper combination and configuration of detection schemes. Huang et al. [13] proposed a risk assessment method that uses a Bayesian network to model the attack propagation process and infers the probabilities of sensors and actuators to be compromised. These probabilities are fed into a stochastic hybrid system (SHS) model to predict the evolution of the physical process being controlled. Then, the security risk is quantified by evaluating the system availability with the SHS model.

Notice that only this last paper adopts formal methodologies. More generally, we are aware of a number of works using formal methods for CPS security, although they apply methods, and most of the time have goals, that are quite different from ours.

Vigo et al. [36] proposed an untimed calculus of broadcasting processes equipped with notions of failed and unwanted communication. They focus on DoS attacks without taking into consideration timing aspects or attack impact. Bodei et al. [2] proposed a different untimed process calculus, IoT-LySa, supporting a control flow analysis that safely approximates the abstract behaviour of IoT systems. Essentially, they track how data spread from sensors to the logics of the network, and how physical data are manipulated. Rocchetto and Tippenhaur [29] introduced a taxonomy of the diverse attacker models proposed for CPS security and outline requirements for generalised attacker models; in [28], the same authors proposed an extended Dolev-Yao attacker model suitable for CPSs. Nigam et al. [24] worked around the notion of timed Dolev-Yao intruder models for cyber-physical security protocols by bounding the number of intruders required for the automated verification of such protocols. Following a tradition in security protocol analysis, they provided an answer to the question: How many intruders are enough for verification and where should they be placed? Lanotte et al. [19] did a static security analysis, based on model-checking, for a non-trivial engineering case study to statically detect a variety of attacks targeting sensors and/or actuators of the system under investigation. Finally, Lanotte et al. [20] defined a hybrid process calculus to model both CPSs and cyber-physical attacks; they defined a threat model for cyber-physical attacks to physical devices and provided a proof methods to assess attack tolerance/vulnerability with respect to a timed trace semantics (no tolerance allowed). They also advocated a timed formalisation of the impact of an attack in terms of the deviation introduced in the runtime behaviour of the system under attack.

Future Work. Recent works [8,9,18,21,22] have shown that bisimulation metrics are suitable for compositional reasoning, as the distance between two complex systems can be often derived in terms of the distance between their components. In this respect, Theorem 2 and Corollary 1 allows us compositional reasonings when computing the impact of attacks on a target system, in terms of the impact on its sub-systems. We believe that this result can be generalised to estimate the impact of parallel attacks of the form $A = A_1 \parallel \dots \parallel A_k$ in terms

of the impacts of each malicious module A_i. As future work, we also intend to adopt our impact metric in more involved languages for *cyber-physical systems and attacks*, such as the language developed in [20], with an explicit representation of physical processes via differential equations or their discrete counterpart, difference equations.

Acknowledgements. We thank the anonymous reviewers for valuable comments.

References

1. Bilis, E.I., Kröger, W., Cen, N.: Performance of electric power systems under physical malicious attacks. IEEE Syst. J. **7**(4), 854–865 (2013)
2. Bodei, C., Degano, P., Ferrari, G., Galletta, L.: Tracing where IoT data are collected and aggregated. Logical Methods Comput. Sci. **13**(3), 1–38 (2017). https://doi.org/10.23638/LMCS-13(3:5)2017
3. van Breugel, F.: On behavioural pseudometrics and closure ordinals. Inf. Process. Lett. **112**(19), 715–718 (2012)
4. Deng, Y., Du, W.: The Kantorovich metric in computer science: a brief survey. ENTCS **253**(3), 73–82 (2009)
5. Desharnais, J., Jagadeesan, R., Gupta, V., Panangaden, P.: The metric analogue of weak bisimulation for probabilistic processes. In: LICS 2002, pp. 413–422. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029849
6. Falliere, N., Murchu, L., Chien, E.: W32.STUXnet Dossier (2011)
7. Focardi, R., Martinelli, F.: A uniform approach for the definition of security properties. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 794–813. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48119-2_44
8. Gebler, D., Larsen, K.G., Tini, S.: Compositional bisimulation metric reasoning with probabilistic process calculi. Logical Meth. Comput. Sci. **12**(4), 1–38 (2016)
9. Gebler, D., Tini, S.: SOS specifications for uniformly continuous operators. J. Comput. Syst. Sci. **92**, 113–151 (2018)
10. Genge, B., Kiss, I., Haller, P.: A system dynamics approach for assessing the impact of cyber attacks on critical infrastructures. IJCIP **10**, 3–17 (2015)
11. Gollmann, D., Gurikov, P., Isakov, A., Krotofil, M., Larsen, J., Winnicki, A.: Cyber-physical systems security: experimental analysis of a vinyl acetate monomer plant. In: Zhou, J., Jones, D. (eds.) ACM CCPS 2015, pp. 1–12. ACM (2015). https://doi.org/10.1145/2732198.2732208
12. Hennessy, M., Regan, T.: A process algebra for timed systems. Inf. Comput. **117**(2), 221–239 (1995)
13. Huang, K., Zhou, C., Tian, Y., Yang, S., Qin, Y.: Assessing the physical impact of cyberattacks on industrial cyber-physical systems. IEEE Trans. Industr. Electron. **65**(10), 8153–8162 (2018)
14. Huang, Y., Cárdenas, A.A., Amin, S., Lin, Z., Tsai, H., Sastry, S.: Understanding the physical and economic consequences of attacks on control systems. IJCIP **2**(3), 73–83 (2009)
15. ICS-CERT: Cyber-Attack Against Ukrainian Critical Infrastructure. https://ics-cert.us-cert.gov/alerts/IR-ALERT-H-16-056-01
16. Keller, R.M.: Formal verification of parallel programs. Commun. ACM **19**, 371–384 (1976)

17. Krotofil, M., Cárdenas, A.A., Larsen, J., Gollmann, D.: Vulnerabilities of cyber-physical systems to stale data - determining the optimal time to launch attacks. IJCIP **7**(4), 213–232 (2014)
18. Lanotte, R., Merro, M.: Semantic analysis of gossip protocols for wireless sensor networks. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 156–170. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23217-6_11
19. Lanotte, R., Merro, M., Munteanu, A.: A modest security analysis of cyber-physical systems: a case study. In: Baier, C., Caires, L. (eds.) FORTE 2018. LNCS, vol. 10854, pp. 58–78. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-319-92612-4
20. Lanotte, R., Merro, M., Muradore, R., Viganò, L.: A formal approach to cyber-physical attacks. In: CSF 2017, pp. 436–450. IEEE Computer Society (2017). https://doi.org/10.1109/CSF.2017.12
21. Lanotte, R., Merro, M., Tini, S.: Compositional weak metrics for group key update. In: Larsen, K.G., Bodlaender, H.L., Raskin, J.F. (eds.) MFCS 2017. LIPIcs, vol. 42, pp. 72:1–72:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017). https://doi.org/10.4230/LIPIcs.MFCS.2017.72
22. Lanotte, R., Merro, M., Tini, S.: A Probabilistic Calculus of Cyber-Physical Systems. CoRR abs/1707.02279 (2017)
23. Lanotte, R., Merro, M., Tini, S.: Towards a formal notion of impact metric for cyber-physical attacks (full version). CoRR abs/1806.10463 (2018)
24. Nigam, V., Talcott, C., Aires Urquiza, A.: Towards the automated verification of cyber-physical security protocols: bounding the number of timed intruders. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9879, pp. 450–470. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45741-3_23
25. Orojloo, H., Azgomi, M.: A method for evaluating the consequence propagation of security attacks in cyber-physical systems. Future Gener. Comput. Syst. **67**, 57–71 (2017)
26. Panangaden, P.: Labelled Markov Processes. Imperial College Press, London (2009)
27. Philippou, A., Lee, I., Sokolsky, O.: Weak bisimulation for probabilistic systems. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 334–349. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44618-4_25
28. Rocchetto, M., Tippenhauer, N.O.: CPDY: extending the Dolev-Yao attacker with physical-layer interactions. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM 2016. LNCS, vol. 10009. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47846-3
29. Rocchetto, M., Tippenhauer, N.O.: On attacker models and profiles for cyber-physical systems. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9879, pp. 427–449. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45741-3_22
30. Segala, R.: Modeling and verification of randomized distributed real-time systems. Ph.D. thesis, MIT (1995)
31. Sgouras, K.I., Birda, A.I., Labridis, D.L.: Cyber attack impact on critical Smart Grid infrastructures. In: IEEE PES ISGT 2014, pp. 1–5. IEEE (2014). https://doi.org/10.1109/ISGT.2014.6816504
32. Slay, J., Miller, M.: Lessons learned from the Maroochy water breach. In: Goetz, E., Shenoi, S. (eds.) ICCIP 2007. IIFIP, vol. 253, pp. 73–82. Springer, Boston (2008). https://doi.org/10.1007/978-0-387-75462-8_6

33. Sridhar, S., Govindarasu, M.: Model-based attack detection and mitigation for automatic generation control. IEEE Trans. Smart Grid **5**(2), 580–591 (2014)
34. Stewart, W.J.: Introduction to the Numerical Solution of Markov Chains. Princeton University Press, Princeton (1994)
35. Urbina, D.I., et al.: Limiting the impact of stealthy attacks on industrial control systems. In: Weippl, E., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016, pp. 1092–1105. ACM (2016) https://doi.org/10.1145/2976749.2978388
36. Vigo, R., Nielson, F., Nielson, H.R.: Broadcast, denial-of-service, and secure communication. In: Johnsen, E.B., Petre, L. (eds.) IFM 2013. LNCS, vol. 7940, pp. 412–427. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38613-8_28
37. Villani, C.: Optimal Transport, Old and New. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71050-9
38. Zacchia Lun, Y., D'Innocenzo, A., Malavolta, I., Di Benedetto, M.D.: Cyber-Physical Systems Security: a Systematic Mapping Study. CoRR abs/1605.09641 (2016)

Task Planning with OMT: An Application to Production Logistics

Francesco Leofante[1,2(✉)], Erika Ábrahám[2], and Armando Tacchella[1]

[1] University of Genoa, Genoa, Italy
[2] RWTH Aachen University, Aachen, Germany
leofante@cs.rwth-aachen.de

Abstract. Task planning is a well-studied problem for which interesting applications exist in production logistics. Planning for such domains requires to take into account not only feasible plans, but also optimality targets, *e.g.*, minimize time, costs or energy consumption. Although there exist several algorithms to compute optimal solutions with formal guarantees, heuristic approaches are typically preferred in practical applications, trading certified solutions for a reduced computational cost. Reverting this trend represents a standing challenge within the domain of task planning at large. In this paper we discuss our experience using Optimization Modulo Theories to synthesize *optimal* plans for multi-robot teams handling production processes within the RoboCup Logistics League. Besides presenting our results, we discuss challenges and possible directions for future development of OMT planning.

1 Introduction

Task planning is the problem of finding a sequence of abstract actions, described at the symbolic level as causal (temporal, spatial) relations in a state transition system, allowing to reach a desired goal state—see, *e.g.*, [11] for a recent account on the subject. In the era of *smart factories*, *i.e.*, responsive, adaptive, connective manufacturing[1], the achievements of task planning research undergo severe challenges when practical applications are in order. Consider a plant that can produce different items on demand. Once a new product request arrives, a robot would probably take the order, assemble it, pack it and prepare it for delivery. With an increasing number of orders, large teams of robots would be needed to keep the business running. Each robot in the team would have to come up with an efficient plan to deliver its order, all while considering what other robots do so as to avoid interferences. Optimality targets like minimizing overall energy consumption or time to delivery, must be taken into account as well.

Task planning in domains as the one just described is an intricate problem that requires reasoning about temporal and ordering constraints efficiently.

[1] https://www2.deloitte.com/insights/us/en/focus/industry-4-0/smart-factory-connected-manufacturing.html.

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 316–325, 2018.
https://doi.org/10.1007/978-3-319-98938-9_18

Although the nature of the constraints involved may vary considerably, the problem remains challenging even in the simplest settings [12]. In order to enable practical application of planning algorithms, relatively fast planners have been developed which rely on heuristics to speed up computations. However, heuristics typically lack performance guarantees, which can be critical to maintain efficiency [3].

In order to study possible solutions to the above problems at a manageable scale, the RoboCup Logistics League (RCLL) [26] has been proposed as a simplified, yet realistic, testbed. Efficient heuristic approaches exist to solve the RCLL [14,25]. However, these methods cannot provide guarantees on the quality of the solutions they produce. To account for this problem, we propose to cast the task planning problem in the framework of Optimization Modulo Theories (OMT): combining symbolic reachability techniques and optimization, OMT solvers [5,32] can be leveraged to generate plans with *formal* performance guarantees by reasoning on expressive models that combine temporal and ordering constraints on tasks and robots.

In the following we discuss our experience using OMT decision procedures to synthesize *optimal* plans within the RoboCup Logistics League. Besides presenting our results, we discuss challenges and possible directions for future development of OMT planning.

2 Satisfiability Modulo Theories and Optimization

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a first-order formula with respect to some decidable theory $\mathcal{T}$. In particular, SMT generalizes Boolean satisfiability (SAT) by adding background theories such as the theory of real numbers, integers, and the theories of data structures.

To decide the satisfiability of an input formula φ in CNF, SMT solvers such as [9,10,23] typically proceed as follows. First a *Boolean abstraction* $abs(\varphi)$ of φ is built by replacing each constraint by a fresh Boolean proposition. A SAT solver searches for a satisfying assignment S for $abs(\varphi)$. If no such assignment exists then the input formula φ is unsatisfiable. Otherwise, the consistency of the assignment in the underlying theory is checked by a *theory solver*. If the constraints are consistent then a satisfying solution (*model*) is found for φ. Otherwise, the theory solver returns a theory lemma φ_E giving an *explanation* for the conflict, *e.g.*, the negated conjunction of some inconsistent input constraints. The explanation is used to refine the Boolean abstraction $abs(\varphi)$ to $abs(\varphi) \wedge abs(\varphi_E)$. These steps are iteratively executed until either a theory-consistent Boolean assignment is found, or no more Boolean satisfying assignments exist.

Standard decision procedures for SMT have been extended with optimization capabilities, leading to Optimization Modulo Theories (OMT). OMT extends SMT solving with optimization procedures to find a variable assignment that defines an optimal value for an objective function f (or a combination of multiple objective functions) under all models of a formula φ. As noted in [31], OMT solvers such as [5,32] typically implement a *linear search* scheme, which can be summarized as follows. Let φ_S be the conjunction of all theory constraints

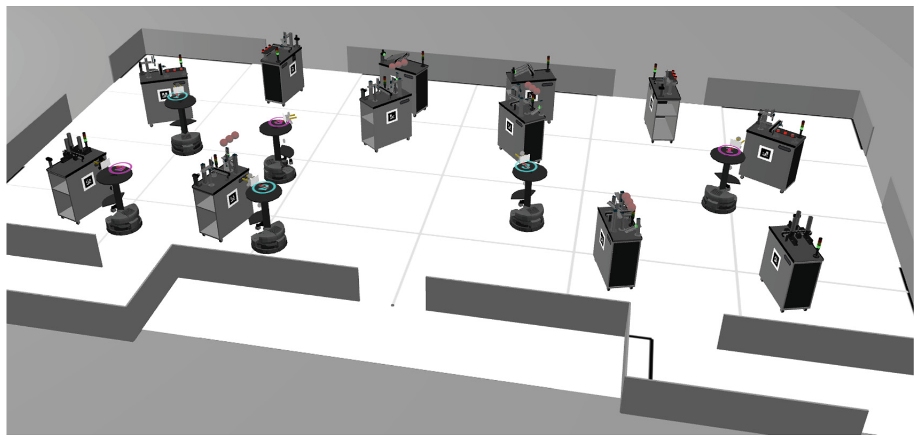


Fig. 1. Simulated RCLL factory environment [24].

that are true under S and the negation of those that are false under S. A local optimum μ for f is computed[2] under the side condition φ_S and φ is updated as

$$\varphi := \varphi \wedge (f \bowtie \mu) \wedge \neg \bigwedge \varphi_S, \quad \bowtie \in \{<, >\}$$

Repeating this procedure until the formula becomes unsatisfiable will lead to an assignment minimizing f under all models of φ.

3 Planning for Autonomous Robots in Smart Factories

In this section we report our experience using OMT to solve planning problems optimally in the RCLL domain. After introducing the domain, we highlight some of the main challenges it presents. In doing so, we wish to distinguish between *(i)* challenges that appear in the RCLL *irrespective* of the approach used for planning and *(ii)* challenges that planning for the RCLL poses to OMT solvers. Finally, we briefly discuss the solution we proposed for OMT planning.

3.1 The RoboCup Logistics League

The RoboCup Logistics League provides a simplified smart factory scenario where two teams of three autonomous robots each compete to handle the logistics of materials to accommodate orders known only at run-time. Competitions take place yearly using a real robotic setup. However, for our experiments we made use of the simulated environment (Fig. 1) developed for the Planning and Execution Competition for Logistics Robots in Simulation[3] [24].

[2] For instance, if f and φ_S are expressed in `QF_LRA`, this can be done with Simplex.
[3] http://www.robocup-logistics.org/sim-comp.

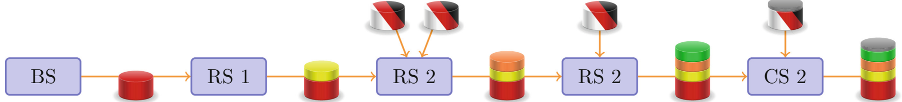


Fig. 2. Example of order configuration for the competition [26,30]. (Color figure online)

Products to be assembled have different complexities and usually require a base, mounting 0 to 3 rings, and a cap as a finishing touch. Bases are available in three different colors, four colors are admissible for rings and two for caps, leading to about 250 different possible combinations. Each order defines which colors are to be used, together with an ordering. An example of a possible configuration is shown in Fig. 2.

Several machines are scattered around the factory shop floor (random placement in each game, positions are announced to the robots). Each of them completes a particular production step such as providing bases, mounting colored rings or caps. There are four types of machines:

- Base Station (BS): acts as dispenser of base elements (one per team).
- Cap Station (CS): mounts a cap as the final step in production on an intermediate product. The CS stores at most one cap at a time (empty initially). To prefill the CS, a base element with a cap must be taken from a shelf in the arena and fed to the machine; the cap is then unmounted and buffered. The cap can then be mounted on the next intermediate product taken to the machine (two CS per team).
- Ring Station (RS): mounts one colored ring (of specific color) onto an intermediate product. Some ring colors require additional tokens: robots will have to feed a RS with a specified number of bases before the required color can be mounted (two RS per team).
- Delivery Station (DS): accepts finished products (one per team).

The objective for autonomous robots is then to transport intermediate products between processing machines and optimize a multistage production cycle of different product variants until delivery of final products.

Orders that denote the products which must be assembled are posted at run-time by an automated referee box and come with a delivery time window, introducing a temporal component that requires quick planning and scheduling.

3.2 RCLL Challenges for Task Planning

Task Planning with Time Windows. Production processes in the RCLL require reasoning over tasks that are subject to temporal constraints often expressed as time windows, *e.g.*, *delivery must happen between minutes 2 and 3.* Time windows increase the complexity of the planning and scheduling problem as they require to reason on both temporal and spatial relations between robots

(and tasks). Furthermore, one might even want to specify additional optimization requirements, such as, *perform delivery taking the least time possible*. The nature of these constraints introduces three strictly related subproblems [28], *i.e.*, *(i)* find an assignment of tasks to robots that optimizes the given objective function, *(ii)* compute a feasible ordering of tasks that results in optimal assignments, and *(iii)* assign times to tasks in a way that optimizes the objective.

Domain Representation. The domain presented by the RCLL is a fairly complex one and efficient reasoning on it can be hard for model-based approaches. To get a feeling of this, consider the variety of product configurations that are allowed in the competition: about 250 configurations are possible! This can clearly represent a problem for state-based approaches, since models explicitly encoding all product variants can quickly become intractable.

Online Execution. Automated production processes like the one proposed by the RCLL are, by their own nature, highly dynamic and subject to unforeseeable changes. The latter are often due to, *e.g.*, temporary robot/machine failures, changes in task definitions or online arrival of new tasks. As all these aspects are modeled in the RCLL, one can easily realize that the execution of plans represents an interesting challenge too. To cater for these aspects, the dynamics that occur during execution must be considered, making *integrated* approaches essential to maintain efficiency.

3.3 Specific Challenges for OMT Solvers

Combinatorial Optimization. Computing optimal task plans with OMT requires the solver to *(i)* search over a finite set of actions so as to *(ii)* optimize an objective function over the arithmetic domain. This means that, although the problem seems to require optimization in the arithmetic domain, it mostly requires efficient combinatorial optimization capabilities. However, solvers do not recognize this and invoke arithmetic optimization resulting in prohibiting runtimes.

Scalability. As the complexity of products required during a game in the RCLL increases, formula encodings can grow rapidly. If large encodings can already become inefficient to solve in SMT, this becomes even more critical in OMT. Abstractions are therefore needed to tame solving complexity, however, currently there are no procedures that can handle iterative abstraction refinement *internally* to the solver. Cimatti et al. started to address the problem in the context of SMT [7,8], however no similar work exists for OMT.

3.4 Our Solution

In a recent series of papers [4,19–22,27] we proposed OMT as an approach to deliver task plans that can meet production requirements (optimally) and withstand deployment in the RCLL. While the approach described in those papers is specifically tailored for the RCLL, we expect that our solution can carry over to

domains with similar structure and features, thus providing the basis for general, yet efficient, synthesis of optimal task plans based on OMT.

To generate optimal plans, we extended standard *Planning as Satisfiability* [16] to enable optimization over reward structures expressed in first-order arithmetic theories in OMT – see [19] for a brief overview of our approach. This idea was applied to solve multi-robot planning problems arising in the RCLL, such as factory shop-floor exploration [21] and planning for production [22].

To cater for the dynamics that occur when plans are executed on concrete systems, we also presented a system that integrates our planning approach into an online execution agent based on CLIPS [25], currently used by the RCLL world champion. A prototypical implementation of this system was presented in [27] and later extended in [22].

4 Further Challenges

Modern solvers achieve impressive performances in several domains and offer a powerful support for planning. Nevertheless, there are several challenges still to be addressed in order to ease the applicability of OMT in planning. In this section we discuss some of these challenges.

Parallelization. Practical efficiency is probably one of the main limitations of current OMT algorithms and tools. Beyond theoretical worst-case complexity results, research on SAT and SMT has shown that problem instances arising from application domains can be tackled successfully in many cases of interest. The research on OMT is currently less mature than its SMT counterpart, therefore improving on this aspect is still an open challenge. We believe that parallelization may offer an interesting path towards attaining practical efficiency in OMT. While parallel SAT solving has been the subject of research in the past [13], the development of parallel SMT solvers is still in its infancy [15], and, to the best of our knowledge, paradigms for parallel OMT have not been considered yet. Carrying over the results obtained in SAT to SMT/OMT is nontrivial, because the role of SAT solver inside SMT/OMT procedure is to *enumerate* assignments and not just to *search* a satisfying one. However, once enumeration of assignments can be successfully parallelized, at least to some extent, also checking their theory consistency and, possibly, finding optimal solutions, can be distributed on several processors. Besides modern multi-core architectures, parallelization could also take advantage of hybrid CPU-GPU architectures, where the GPU part can substantially speed up numerical computations as it happens in other AI fields like training of deep neural networks—see, *e.g.*, [17].

Develop New Encodings. Our experiments with different encodings of planning problems into OMT indicate that considerable progress can be made by considering novel kinds of encodings and relaxations. Beyond computational concerns, new relaxations can be of great interest from a representational standpoint. One key challenge relates to finding encodings which *generalize* well to several problem domains. While the improvements that we obtained working on the RCLL

domain are mostly specific and tailored to a specific scenario, some choices, *e.g.*, state-based vs. action-based encoding, should be weighted across several domains to understand their effectiveness at large. Currently, to the extent of our knowledge, the proposal of OMT-based planning has been tried and tested successfully in our work only, whereas general-purpose SMT-based planners have been proposed already—see, *e.g.*, [6]. In both cases, the main issue is to deal effectively with the combinatorial aspects of the domain which could force the SAT engine underlying the SMT/OMT solver to perform "brute-force search" in the space of possible assignments without a clue. In order to reduce this phenomenon, some domain knowledge must be incorporated in the encoding so as to allow the SAT solver to learn suitable propositional constrains by interacting with the theory solver. In addition to this, specialized SAT heuristics could be devised to minimize the chance of exploring parts of the propositional search space which are trivially unfruitful once the underlying theory is taken into account.

Accounting for Uncertainty. Planning for dynamic environments such as smart factories often requires to include some degree of uncertainty in the models used. While practically efficient techniques to reason about uncertain models have been proposed in task planning—see, *e.g.*, Part V of [11]—and formal verification—see, *e.g.*, [2]—the question remains how to encode uncertainty efficiently in SMT/OMT, or to extend existing decision procedures to incorporate uncertainty. The issue is nontrivial because many of the above mentioned probabilistic techniques and tools rely on state-based representations which quickly become intractable in realistic production logistics domains. A symbolic approach, wherein the specification of states and transitions is left implicit like in probabilistic programs, would be suitable for realistically-sized domains as well. SMT solvers have been successfully used both in planning and formal verification as engines in symbolic planners and model checkers, but this is not the case of OMT solvers and, to the best of our knowledge, there is no provision for optimization in current state-of-the-art tools.

Integrated Task and Motion Planning. Task planning tries to answer the question of *what* can be done to achieve a given objective. In the RCLL, this would correspond to, *e.g.*, *what actions do robots need to perform to deliver the requested product?* However, as highlighted in Sect. 3, planning for robotics requires more than just saying what needs to be done. In such domains, planning needs to be integrated with other deliberation functions such as *how* to perform actions (and monitor their progress). In our previous work, we decoupled this problem by relying on an execution and monitoring agent *external* to the planning process. This however may lead to plan failures if, *e.g.*, a robot tries to move toward a target location to perform a task but its kinematics does not allow such motion. A tighter integration between *task* constraints and *motion* constraints is needed in most relevant robotics applications where any planning is needed, and such need has been recognized for a long time—see, *e.g.*, [29]. However, it was not until recent times that research in motion planning achieved efficient algorithms and tools—see, *e.g.*, [18]—that can be combined over a wide spectrum of methods and

implementations in task planning. Harvesting motion planning techniques and incorporating them into SMT/OMT based approaches requires matching, mostly discrete, specifications of task planning domains with the, mostly continuous, specification of motion planning constraints. A suitable semantic "bridge" must be found between the two sides, one that is akin to hybdrid systems models where discrete control modes are endowed with continuous dynamics.

Explainable Planning. The problem of generating explanations for decisions taken by autonomous systems is very pressing, with many initiative being launched to foster research in this field [1]. The need for explanations is even amplified when AI technologies are employed in safety-critical scenarios involving, *e.g.*, the interplay of human and robotic workers. In [22] we started looking into this problem, as we believe OMT-based synthesis builds on techniques that have the potential to ease explaining, *e.g.*, by generating *unsat cores*. However, presenting solver outputs in a human-readable way that could facilitate understanding of the underlying decision process remains an open challenge.

5 Conclusion

In this paper we briefly reported on our experiences using Optimization Modulo Theories to solve task planning problems in logistics. Given the broad nature of the topic, we focused our effort on problems stemming from the RoboCup Logistics League, a well-known benchmark in planning and robotics. Despite highly encouraging results, efforts are still needed to increase applicability and scalability of OMT technologies in planning. With this work we hope to increase the visibility of OMT techniques and tools in planning so as to intensify the developments in this relevant research area.

References

1. EXplainable AI Planning (2018). http://icaps18.icaps-conference.org/xaip/
2. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
3. Bensalem, S., Havelund, K., Orlandini, A.: Verification and validation meet planning and scheduling. STTT **16**(1), 1–12 (2014)
4. Bit-Monnot, A., Leofante, F., Pulina, L., Ábrahám, E., Tacchella, A.: SMarTplan: a task planner for smart factories. CoRR abs/1806.07135 (2018)
5. Bjørner, N., Phan, A.-D., Fleckenstein, L.: νz - An optimizing SMT solver. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 194–199. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_14
6. Cashmore, M., Fox, M., Long, D., Magazzeni, D.: A compilation of the full PDDL+ language into SMT. In: Proceedings of ICAPS 2016, pp. 79–87 (2016)
7. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Invariant checking of NRA transition systems via incremental reduction to LRA with EUF. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 58–75. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_4

8. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Satisfiability modulo transcendental functions via incremental linearization. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 95–113. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_7
9. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
10. Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E.: SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 360–368. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_26
11. Ghallab, M., Nau, D.S., Traverso, P.: Automated Planning and Acting. Cambridge University Press, Cambridge (2016)
12. Gini, M.L.: Multi-robot allocation of tasks with temporal and ordering constraints. In: Proceedings of AAAI 2018, pp. 4863–4869 (2017)
13. Hamadi, Y., Wintersteiger, C.M.: Seven challenges in parallel SAT solving. AI Mag. **34**(2), 99–106 (2013)
14. Hofmann, T., Niemueller, T., Claßen, J., Lakemeyer, G.: Continual planning in Golog. In: Proceeding of AAAI 2016, pp. 3346–3353 (2016)
15. Hyvärinen, A.E.J., Wintersteiger, C.M.: Parallel satisfiability modulo theories. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning, pp. 141–178. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-63516-3_5
16. Kautz, H.A., Selman, B.: Planning as satisfiability. In: ECAI, pp. 359–363 (1992)
17. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Proceedings of NIPS 2012, pp. 1097–1105 (2012)
18. LaValle, S.M.: Planning Algorithms. Cambridge University Press, Cambridge (2006)
19. Leofante, F.: Guaranteed plans for multi-robot systems via optimization modulo theories. In: Proceedings of AAAI 2018 (2018)
20. Leofante, F.: Optimal multi-robot task planning: from synthesis to execution (and back). In: Proceeding of IJCAI 2018 (2018, to appear)
21. Leofante, F., Ábrahám, E., Niemueller, T., Lakemeyer, G., Tacchella, A.: On the synthesis of guaranteed-quality plans for robot fleets in logistics scenarios via optimization modulo theories. In: Proceedings of IRI 2017, pp. 403–410 (2017)
22. Leofante, F., Ábrahám, E., Niemueller, T., Lakemeyer, G., Tacchella, A.: Integrated Synthesis and Execution of Optimal Plans for Multi-Robot Systems in Logistics. Information Systems Frontiers, pp. 1–21. Springer, New York (2018). https://doi.org/10.1007/s10796-018-9858-3
23. 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). https://doi.org/10.1007/978-3-540-78800-3_24
24. Niemueller, T., Karpas, E., Vaquero, T., Timmons, E.: Planning competition for logistics robots in simulation. In: Proceeding of PlanRob@ICAPS 2016 (2016)
25. Niemueller, T., Lakemeyer, G., Ferrein, A.: Incremental task-level reasoning in a competitive factory automation scenario. In: AAAI Spring Symposium - Designing Intelligent Robots: Reintegrating AI (2013)
26. Niemueller, T., Lakemeyer, G., Ferrein, A.: The RoboCup Logistics League as a benchmark for planning in robotics. In: Proceeding of PlanRob@ICAPS 2015 (2015)

27. Niemueller, T., Lakemeyer, G., Leofante, F., Ábrahám, E.: Towards CLIPS-based task execution and monitoring with SMT-based decision optimization. In: Proceedings of PlanRob@ICAPS 2017 (2017)
28. Nunes, E., Manner, M.D., Mitiche, H., Gini, M.L.: A taxonomy for task allocation problems with temporal and ordering constraints. Robot. Auton. Syst. **90**, 55–70 (2017)
29. Popplestone, R., Ambler, A., Bellos, I.: RAPT: a language for describing assemblies. Ind. Robot Int. J. **5**(3), 131–137 (1978)
30. RCLL Technical Committee: RoboCup Logistics League – Rules and regulations 2017 (2017)
31. Sebastiani, R., Tomasi, S.: Optimization modulo theories with linear rational costs. ACM Trans. Comput. Log. **16**(2), 12:1–12:43 (2015)
32. Sebastiani, R., Trentin, P.: OptiMathSAT: a tool for optimization modulo theories. In: Proceedings of CAV 2015, pp. 447–454 (2015)

Branching Temporal Logic of Calls and Returns for Pushdown Systems

Huu-Vu Nguyen[1(✉)] and Tayssir Touili[2(✉)]

[1] LIPN, CNRS and University Paris 13, Villetaneuse, France
nguyen@lipn.univ-paris13.fr
[2] CNRS, LIPN and University Paris 13, Villetaneuse, France
touili@lipn.univ-paris13.fr

Abstract. Pushdown Systems (PDSs) are a natural model for sequential programs with (recursive) procedure calls. In this work, we define the Branching temporal logic of CAlls and RETurns (BCARET) that allows to write branching temporal formulas while taking into account the matching between calls and returns. We consider the model-checking problem of PDSs against BCARET formulas with "standard" valuations (where an atomic proposition holds at a configuration c or not depends only on the control state of c, not on its stack) as well as regular valuations (where the set of configurations in which an atomic proposition holds is regular). We show that these problems can be effectively solved by a reduction to the emptiness problem of Alternating Büchi Pushdown Systems. We show that our results can be applied for malware detection.

1 Introduction

Pushdown Systems (PDSs) are a natural model for sequential programs with (recursive) procedure calls. Thus, it is very important to have model-checking algorithms for PDSs. A lot of work focuses on proposing verification algorithms for PDSs, e.g, for both linear temporal logic (LTL and its extensions) [6,9–12,17] and branching temporal logic (CTL and its extensions) [6–8,15,18]. However, LTL and CTL are not always adequate to specify properties. Indeed, some properties need to talk about matching between calls and returns. Thus, CARET (a temporal logic of calls and returns) was introduced by Alur et al. [5]. This logic allows to write linear temporal logic formulas while taking into account matching of calls and returns. Later, VP-μ (also named NT-μ in other works of the same authors) [2–4], a branching-time temporal logic that allows to talk about matching between calls and returns, was introduced. VP-μ can be seen as an extension of the modal μ-calculus which allows to talk about matching of calls and returns.

In [2], the authors proposed an algorithm to model-check VP-μ formulas for Recursive State Machines (RSMs) [1]. RSMs can be seen as a natural model to represent sequential programs with (recursive) procedure calls. Each procedure is modelled as a module. The invocation to a procedure is modelled as a *call*

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 326–345, 2018.
https://doi.org/10.1007/978-3-319-98938-9_19

node; the return from a module corresponds to a *ret* node; and the remaining statements are considered as internal nodes in the RSMs. Thus, RSMs are a good formalism to model sequential programs written in structured programming languages like C or Java. However, they become non suitable for modelling binary or assembly programs; since, in these programs, explicit push and pop instructions can occur. This makes impossible the use of RSMs to model assembly programs and binary codes directly (whereas Pushdown Systems can model binary codes in a natural way [16]). Model checking binary and assembly programs is very important. Indeed, sometimes, only the binary code is available. Moreover, malicious programs are often executables, i.e., binary codes. Thus, it is very important to be able to model check binary and assembly programs against branching-time formulas with matchings between calls and returns. One can argue that from a binary/assembly program, one can compute a PDS as described in [16] and then apply the translation in [1] to obtain a RSM and then apply the VP-μ model-checking algorithm of [2] on this RSM. However, by doing so, we loose the explicit manipulation of the program's stack. Explicit push and pop instructions are not represented in a natural way anymore, and the stack of the RSM does not correspond to the stack of the assembly program anymore. Thus, it is not possible to state intuitive formulas that correspond to properties of the program's behaviors on the obtained RSM. Especially, when these formulas talk about the content of the program's stack. Thus, it is very important to have a *direct* algorithm for model-checking a branching-time temporal logic with matching of calls and returns for PDSs.

However, VP-μ is a heavy formalism that can't be used by novice users. Indeed, VP-μ can be seen as an extension of the modal μ calculus with several modalities $\langle loc \rangle$, $[loc]$, $\langle call \rangle$, $[call]$, $\langle ret \rangle$, $[ret]$ that allow to distinguish between calls, returns, and other statements (neither calls nor returns). Writing a simple specification in VP-μ is complicated. For example, the following simple property stating that "the configuration e can be reached in the same procedural context as the current configuration" can be described (as shown in [2]) by the complex VP-μ formula $\varphi_2' = \mu X(e \vee \langle loc \rangle X \vee \langle call \rangle \varphi_3' \{X\})$ where $\varphi_3' = \mu Y(\langle ret \rangle R_1 \vee \langle loc \rangle Y \vee \langle call \rangle Y\{Y\})$. Thus, we need to define a more intuitive branching-time temporal logic (in the style of CTL) that allow to talk naturally and intuitively about matching calls and returns.

Therefore, we define in this work the Branching temporal logic of CAlls and RETurns BCARET. BCARET can be seen as an extension of CTL with operators that allow to talk about matchings between calls and returns. Using BCARET, the above reachability property can be described in a simple way by the formula $EF^a e$ where EF^a is a BCARET operator that means "there exists a run on which eventually in the future in the same procedural context". We consider the model-checking problem of PDSs against BCARET formulas with "standard" valuations (where an atomic proposition holds at a configuration c or not depends only on the control state of c, not on its stack) as well as regular valuations (where the set of configurations in which an atomic proposition holds is a regular set of configurations). We show that these problems can be effectively

solved by a reduction to the emptiness problem of Alternating Büchi Pushdown Systems (ABPDSs). The latter problem can be solved effectively in [15]. Note that the regular valuation case cannot be solved by translating the PDSs to RSMs since as said previously, by doing the translation of PDSs to obtain RSMs, we loose the structure of the program's stack.

The rest of the paper is organized as follows. In Sect. 2, we define Labelled Pushdown Systems. In Sect. 3, we define the logic BCARET. Section 4 presents applications of BCARET in specifying malicious behaviours. Our algorithm to reduce BCARET model-checking to the membership problem of ABPDSs is presented in Sect. 5. Section 6 discusses the model-checking problem for PDSs against BCARET formulas with regular valuations. Finally, we conclude in Sect. 7.

2 Pushdown Systems: A Model for Sequential Programs

Pushdown systems is a natural model that was extensively used to model sequential programs. Translations from sequential programs to PDSs can be found e.g. in [14]. As will be discussed in the next section, to precisely describe malicious behaviors as well as context-related properties, we need to keep track of the call and return actions in each path. Thus, as done in [13], we adapt the PDS model in order to record whether a rule of a PDS corresponds to a *call*, a *return*, or another instruction. We call this model a *Labelled Pushdown System*. We also extend the notion of *run* in order to take into account matching returns of calls.

Definition 1. *A Labelled Pushdown System (PDS) $\mathcal{P}$ is a tuple $(P, \Gamma, \Delta, \sharp)$, where P is a finite set of control locations, Γ is a finite set of stack alphabet, $\sharp \notin \Gamma$ is a bottom stack symbol and Δ is a finite subset of $((P \times \Gamma) \times (P \times \Gamma^*) \times \{call, ret, int\})$. If $((p,\gamma),(q,\omega),t) \in \Delta$ ($t \in \{call, ret, int\}$), we also write $\langle p,\gamma\rangle \xrightarrow{t} \langle q,\omega\rangle \in \Delta$. Rules of Δ are of the following form, where $p \in P, q \in P, \gamma, \gamma_1, \gamma_2 \in \Gamma$, and $\omega \in \Gamma^*$:*

- *(r_1): $\langle p,\gamma\rangle \xrightarrow{call} \langle q,\gamma_1\gamma_2\rangle$*
- *(r_2): $\langle p,\gamma\rangle \xrightarrow{ret} \langle q,\epsilon\rangle$*
- *(r_3): $\langle p,\gamma\rangle \xrightarrow{int} \langle q,\omega\rangle$*

Intuitively, a rule of the form $\langle p,\gamma\rangle \xrightarrow{call} \langle q,\gamma_1\gamma_2\rangle$ corresponds to a call statement. Such a rule usually models a statement of the form $\gamma \xrightarrow{call\ \ proc} \gamma_2$. In this rule, γ is the control point of the program where the function call is made, γ_1 is the entry point of the called procedure, and γ_2 is the return point of the call. A rule r_2 models a return, whereas a rule r_3 corresponds to a *simple* statement (neither a call nor a return). A configuration of $\mathcal{P}$ is a pair $\langle p,\omega\rangle$, where p is a control location and $\omega \in \Gamma^*$ is the stack content. For technical reasons, we suppose w.l.o.g. that the bottom stack symbol $\sharp$ is never popped from the stack, i.e., there is no rule in the form $\langle p,\sharp\rangle \xrightarrow{t} \langle q,\omega\rangle \in \Delta$ ($t \in \{call, ret, int\}$). $\mathcal{P}$ defines

a transition relation $\Rightarrow_{\mathcal{P}}$ ($t \in \{call, ret, int\}$)as follows: If $\langle p, \gamma \rangle \xrightarrow{t} \langle q, \omega \rangle$, then for every $\omega' \in \Gamma^*$, $\langle p, \gamma\omega' \rangle \Rightarrow_{\mathcal{P}} \langle q, \omega\omega' \rangle$. In other words, $\langle q, \omega\omega' \rangle$ is an immediate successor of $\langle p, \gamma\omega' \rangle$. Let $\overset{*}{\Rightarrow}_{\mathcal{P}}$ be the reflexive and transitive closure of $\Rightarrow_{\mathcal{P}}$.

A run of $\mathcal{P}$ from $\langle p_0, \omega_0 \rangle$ is a sequence $\langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle \langle p_2, \omega_2 \rangle$... where $\langle p_i, \omega_i \rangle \in P \times \Gamma^*$ s.t. for every $i \geq 0, \langle p_i, \omega_i \rangle \Rightarrow_{\mathcal{P}} \langle p_{i+1}, \omega_{i+1} \rangle$. Given a configuration $\langle p, \omega \rangle$, let $Traces(\langle p, \omega \rangle)$ be the set of all possible runs starting from $\langle p, \omega \rangle$.

2.1 Global and Abstract Successors

Let $\pi = \langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle$... be a run starting from $\langle p_0, \omega_0 \rangle$. Over π, two kinds of successors are defined for every position $\langle p_i, \omega_i \rangle$:

- *global-successor*: The global-successor of $\langle p_i, \omega_i \rangle$ is $\langle p_{i+1}, \omega_{i+1} \rangle$ where $\langle p_{i+1}, \omega_{i+1} \rangle$ is an immediate successor of $\langle p_i, \omega_i \rangle$.
- *abstract-successor*: The abstract-successor of $\langle p_i, \omega_i \rangle$ is determined as follows:
  - If $\langle p_i, \omega_i \rangle \Rightarrow_{\mathcal{P}} \langle p_{i+1}, \omega_{i+1} \rangle$ corresponds to a call statement, there are two cases: (1) if $\langle p_i, \omega_i \rangle$ has $\langle p_k, \omega_k \rangle$ as a corresponding return-point in π, then, the abstract successor of $\langle p_i, \omega_i \rangle$ is $\langle p_k, \omega_k \rangle$; (2) if $\langle p_i, \omega_i \rangle$ does not have any corresponding return-point in π, then, the abstract successor of $\langle p_i, \omega_i \rangle$ is $\perp$.
  - If $\langle p_i, \omega_i \rangle \Rightarrow_{\mathcal{P}} \langle p_{i+1}, \omega_{i+1} \rangle$ corresponds to a *simple* statement, the abstract successor of $\langle p_i, \omega_i \rangle$ is $\langle p_{i+1}, \omega_{i+1} \rangle$.
  - If $\langle p_i, \omega_i \rangle \Rightarrow_{\mathcal{P}} \langle p_{i+1}, \omega_{i+1} \rangle$ corresponds to a return statement, the abstract successor of $\langle p_i, \omega_i \rangle$ is defined as $\perp$.

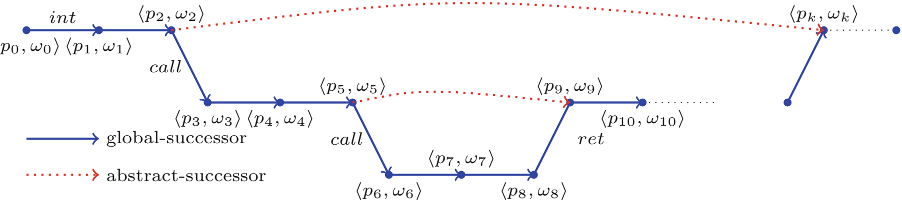


Fig. 1. Two kinds of successors on a run

For example, in Fig. 1:

- The global-successors of $\langle p_1, \omega_1 \rangle$ and $\langle p_2, \omega_2 \rangle$ are $\langle p_2, \omega_2 \rangle$ and $\langle p_3, \omega_3 \rangle$ respectively.
- The abstract-successors of $\langle p_2, \omega_2 \rangle$ and $\langle p_5, \omega_5 \rangle$ are $\langle p_k, \omega_k \rangle$ and $\langle p_9, \omega_9 \rangle$ respectively.

Let $\langle p, \omega \rangle$ be a configuration of a PDS $\mathcal{P}$. A configuration $\langle p', \omega' \rangle$ is defined as a global-successor of $\langle p, \omega \rangle$ iff $\langle p', \omega' \rangle$ is a global-successor of $\langle p, \omega \rangle$ over a run $\pi \in Traces(\langle p, \omega \rangle)$. Similarly, a configuration $\langle p', \omega' \rangle$ is defined as an abstract-successor of $\langle p, \omega \rangle$ iff $\langle p', \omega' \rangle$ is an abstract-successor of $\langle p, \omega \rangle$ over a run $\pi \in Traces(\langle p, \omega \rangle)$.

A *global-path* of $\mathcal{P}$ from $\langle p_0, \omega_0 \rangle$ is a sequence $\langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle \langle p_2, \omega_2 \rangle$... where $\langle p_i, \omega_i \rangle \in P \times \Gamma^*$ s.t. for every $i \geq 0$, $\langle p_{i+1}, \omega_{i+1} \rangle$ is a global-successor of $\langle p_i, \omega_i \rangle$. Similarly, an *abstract-path* of $\mathcal{P}$ from $\langle p_0, \omega_0 \rangle$ is a sequence $\langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle \langle p_2, \omega_2 \rangle$... where $\langle p_i, \omega_i \rangle \in P \times \Gamma^*$ s.t. for every $i \geq 0$, $\langle p_{i+1}, \omega_{i+1} \rangle$ is an abstract-successor of $\langle p_i, \omega_i \rangle$. For instance, in Fig. 1, $\langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle \langle p_2, \omega_2 \rangle \langle p_3, \omega_3 \rangle \langle p_4, \omega_4 \rangle \langle p_5, \omega_5 \rangle$... is a global-path, while $\langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle \langle p_2, \omega_2 \rangle \langle p_k, \omega_k \rangle$... is an abstract-path.

2.2 Multi Automata

Definition 2. *[6] Let $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$ be a PDS. A $\mathcal{P}$-Multi-Automaton (MA for short) is a tuple $\mathcal{A} = (Q, \Gamma, \delta, I, Q_f)$, where Q is a finite set of states, $\delta \subseteq Q \times \Gamma \times Q$ is a finite set of transition rules, $I = P \subseteq Q$ is a set of initial states, $Q_f \subseteq Q$ is a set of final states.*

The transition relation $\rightarrow_\delta \subseteq Q \times \Gamma^* \times Q$ is defined as follows:

- $q \xrightarrow{\epsilon}_\delta q$ for every $q \in Q$
- $q \xrightarrow{\gamma}_\delta q'$ if $(q, \gamma, q') \in \delta$
- if $q \xrightarrow{\omega}_\delta q'$ and $q' \xrightarrow{\gamma}_\delta q''$, then, $q \xrightarrow{\omega\gamma}_\delta q''$

$\mathcal{A}$ recognizes a configuration $\langle p, \omega \rangle$ where $p \in P$, $\omega \in \Gamma^*$ iff $p \xrightarrow{\omega}_\delta q$ for some $q \in Q_f$. The language of $\mathcal{A}$, $L(\mathcal{A})$, is the set of all configurations which are recognized by $\mathcal{A}$. A set of configurations is *regular* if it is recognized by some Multi-Automaton.

3 Branching Temporal Logic of Calls and Returns - BCARET

In this section, we define the Branching temporal logic of CAlls and RETurns BCARET. For technical reasons, we assume w.l.o.g. that BCARET formulas are given in positive normal form, i.e. negations are applied only to atomic propositions. To do that, we use the release operator R as a dual of the until operator U.

Definition 3. *Syntax of BCARET*
Let AP be a finite set of atomic propositions, a BCARET formula φ is defined as follows, where $b \in \{g, a\}$, $e \in AP$:

$$\varphi ::= true \mid false \mid e \mid \neg e \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid EX^b\varphi \mid AX^b\varphi \mid E[\varphi U^b \varphi] \mid A[\varphi U^b \varphi] \mid E[\varphi R^b \varphi] \mid A[\varphi R^b \varphi]$$

Let $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$ be a PDS, $\lambda : AP \to 2^{P \times \Gamma^*}$ be a labelling function that assigns to each atomic proposition $e \in AP$ a set of configurations of $\mathcal{P}$. The satisfiability relation of a BCARET formula φ at a configuration $\langle p_0, \omega_0 \rangle$ w.r.t. the labelling function λ, denoted by $\langle p_0, \omega_0 \rangle \models_\lambda \varphi$, is defined inductively as follows:

- $\langle p_0, \omega_0 \rangle \models_\lambda true$ for every $\langle p_0, \omega_0 \rangle$
- $\langle p_0, \omega_0 \rangle \not\models_\lambda false$ for every $\langle p_0, \omega_0 \rangle$
- $\langle p_0, \omega_0 \rangle \models_\lambda e$ $(e \in AP)$ iff $\langle p_0, \omega_0 \rangle \in \lambda(e)$
- $\langle p_0, \omega_0 \rangle \models_\lambda \neg e$ $(e \in AP)$ iff $\langle p_0, \omega_0 \rangle \notin \lambda(e)$
- $\langle p_0, \omega_0 \rangle \models_\lambda \varphi_1 \vee \varphi_2$ iff $(\langle p_0, \omega_0 \rangle \models_\lambda \varphi_1$ or $\langle p_0, \omega_0 \rangle \models_\lambda \varphi_2)$
- $\langle p_0, \omega_0 \rangle \models_\lambda \varphi_1 \wedge \varphi_2$ iff $(\langle p_0, \omega_0 \rangle \models_\lambda \varphi_1$ and $\langle p_0, \omega_0 \rangle \models_\lambda \varphi_2)$
- $\langle p_0, \omega_0 \rangle \models_\lambda EX^g\varphi$ iff there exists a global-successor $\langle p', \omega' \rangle$ of $\langle p_0, \omega_0 \rangle$ such that $\langle p', \omega' \rangle \models_\lambda \varphi$
- $\langle p_0, \omega_0 \rangle \models_\lambda AX^g\varphi$ iff $\langle p', \omega' \rangle \models_\lambda \varphi$ for every global-successor $\langle p', \omega' \rangle$ of $\langle p_0, \omega_0 \rangle$
- $\langle p_0, \omega_0 \rangle \models_\lambda E[\varphi_1 U^g \varphi_2]$ iff there exists a global-path $\pi = \langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle \langle p_2, \omega_2 \rangle ...$ of $\mathcal{P}$ starting from $\langle p_0, \omega_0 \rangle$ s.t. $\exists i \geq 0$, $\langle p_i, \omega_i \rangle \models_\lambda \varphi_2$ and for every $0 \leq j < i$, $\langle p_j, \omega_j \rangle \models_\lambda \varphi_1$
- $\langle p_0, \omega_0 \rangle \models_\lambda A[\varphi_1 U^g \varphi_2]$ iff for every global-path $\pi = \langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle \langle p_2, \omega_2 \rangle ...$ of $\mathcal{P}$ starting from $\langle p_0, \omega_0 \rangle$, $\exists i \geq 0$, $\langle p_i, \omega_i \rangle \models_\lambda \varphi_2$ and for every $0 \leq j < i$, $\langle p_j, \omega_j \rangle \models_\lambda \varphi_1$
- $\langle p_0, \omega_0 \rangle \models_\lambda E[\varphi_1 R^g \varphi_2]$ iff there exists a global-path $\pi = \langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle \langle p_2, \omega_2 \rangle ...$ of $\mathcal{P}$ starting from $\langle p_0, \omega_0 \rangle$ s.t. for every $i \geq 0$, if $\langle p_i, \omega_i \rangle \not\models_\lambda \varphi_2$ then there exists $0 \leq j < i$ s.t. $\langle p_j, \omega_j \rangle \models_\lambda \varphi_1$
- $\langle p_0, \omega_0 \rangle \models_\lambda A[\varphi_1 R^g \varphi_2]$ iff for every global-path $\pi = \langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle \langle p_2, \omega_2 \rangle ...$ of $\mathcal{P}$ starting from $\langle p_0, \omega_0 \rangle$, for every $i \geq 0$, if $\langle p_i, \omega_i \rangle \not\models_\lambda \varphi_2$ then there exists $0 \leq j < i$ s.t. $\langle p_j, \omega_j \rangle \models_\lambda \varphi_1$
- $\langle p_0, \omega_0 \rangle \models_\lambda EX^a\varphi$ iff there exists an abstract-successor $\langle p', \omega' \rangle$ of $\langle p_0, \omega_0 \rangle$ such that $\langle p', \omega' \rangle \models_\lambda \varphi$
- $\langle p_0, \omega_0 \rangle \models_\lambda AX^a\varphi$ iff $\langle p', \omega' \rangle \models_\lambda \varphi$ for every abstract-successor $\langle p', \omega' \rangle$ of $\langle p_0, \omega_0 \rangle$
- $\langle p_0, \omega_0 \rangle \models_\lambda E[\varphi_1 U^a \varphi_2]$ iff there exists an abstract-path $\pi = \langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle \langle p_2, \omega_2 \rangle ...$ of $\mathcal{P}$ starting from $\langle p_0, \omega_0 \rangle$ s.t. $\exists i \geq 0$, $\langle p_i, \omega_i \rangle \models_\lambda \varphi_2$ and for every $0 \leq j < i$, $\langle p_j, \omega_j \rangle \models_\lambda \varphi_1$
- $\langle p_0, \omega_0 \rangle \models_\lambda A[\varphi_1 U^a \varphi_2]$ iff for every abstract-path $\pi = \langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle \langle p_2, \omega_2 \rangle ...$ of $\mathcal{P}$, $\exists i \geq 0$, $\langle p_i, \omega_i \rangle \models_\lambda \varphi_2$ and for every $0 \leq j < i$, $\langle p_j, \omega_j \rangle \models_\lambda \varphi_1$
- $\langle p_0, \omega_0 \rangle \models_\lambda E[\varphi_1 R^a \varphi_2]$ iff there exists an abstract-path $\pi = \langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle \langle p_2, \omega_2 \rangle ...$ of $\mathcal{P}$ starting from $\langle p_0, \omega_0 \rangle$ s.t. for every $i \geq 0$, if $\langle p_i, \omega_i \rangle \not\models_\lambda \varphi_2$ then there exists $0 \leq j < i$ s.t. $\langle p_j, \omega_j \rangle \models_\lambda \varphi_1$
- $\langle p_0, \omega_0 \rangle \models_\lambda A[\varphi_1 R^a \varphi_2]$ iff for every abstract-path $\pi = \langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle \langle p_2, \omega_2 \rangle ...$ of $\mathcal{P}$ starting from $\langle p_0, \omega_0 \rangle$, for every $i \geq 0$, if $\langle p_i, \omega_i \rangle \not\models_\lambda \varphi_2$ then there exists $0 \leq j < i$ s.t. $\langle p_j, \omega_j \rangle \models_\lambda \varphi_1$

Other BCARET operators can be expressed by the above operators: $EF^g\varphi = E[true\ U^g\varphi]$, $EF^a\varphi = E[true\ U^a\varphi]$, $AF^g\varphi = A[true\ U^g\varphi]$, $AF^a\varphi = A[trueU^a\varphi]$,...

Closure. Given a BCARET formula φ, the closure $Cl(\varphi)$ is the set of all subformulae of φ, including φ.

Regular Valuations. We talk about regular valuations when for every $e \in AP$, $\lambda(e)$ is a regular language.

Remark 1. CTL can be seen as the subclass of BCARET where the operators $EX^a\varphi, AX^a\varphi, E[\varphi U^a \varphi], A[\varphi U^a \varphi], E[\varphi R^a \varphi], A[\varphi R^a \varphi]$ are not considered.

4 Application

In this section, we show how BCARET can be used to describe branching-time malicious behaviors.

Spyware Behavior. The typical behaviour of a spyware is hunting for personal information (emails, bank account information,...) on local drives by searching files matching certain conditions. To do that, it has to search directories of the host to look for interesting files whose names match a specific condition. When a file is found, the spyware will invoke a payload to steal the information, then continue looking for the remaining matching files. When a folder is found, it will enter the folder path and continue scanning that folder recursively. To achieve this behavior, the spyware first calls the API function $FindFirstFileA$ to search for the first matching file in a given folder path. After that, it has to check whether the call to the API function $FindFirstFileA$ succeeds or not. If the function call fails, the spyware will call the function $GetLastError$. Otherwise, if the function call is successful, $FindFirstFileA$ will return a search handle h. There are two possibilities in this case. If the returned result is a folder, it will call the API function $FindFirstFileA$ again to search for matching results in the found folder. If the returned result is a file, it will call the API function $FindNextFileA$ using h as first parameter to look for the remaining matching files. This behavior cannot be expressed by LTL or CTL because it requires to express that the return value of the function $FindFirstFileA$ should be used as input to the API function $FindNextFileA$. It cannot be described by CARET neither (because this is a branching-time property). Using BCARET, the above behavior can be expressed by the following formula:

$$\varphi_{sb} = \bigvee_{d \in D} EF^g \Bigg(call(FindFirstFileA) \wedge EX^a(eax = d) \wedge AF^a \Big(call(GetLastError) \vee call(FindFirstFileA) \vee \Big(call(FindNextFileA) \wedge d\Gamma^* \Big) \Big) \Bigg)$$

where the $\bigvee$ is taken over all possible memory addresses d which contain the values of search handles h in the program, EX^a is a BCARET operator that

means "next in some run, in the same procedural context"; EF^g is the standard CTL EF operator (eventually in some run), while AF^a is a BCARET operator that means "eventually in all runs, in the same procedural context".

In binary codes and assembly programs, the return value of an API function is put in the register eax. Thus, the return value of $FindFirstFileA$ is the value of eax at its corresponding return-point. Then, the subformula (call(FindFirstFileA) $\wedge$ $EX^a(eax = d)$) states that there is a call to the API $FindFirstFileA$ and the return value of this function is d (the abstract successor of a call is its corresponding return-point). When *FindNextFileA* is invoked, it requires a search handle as parameter and this search handle must be put on top of the program stack (since parameters are passed through the stack in assembly). The requirement that d is on top of the program stack is expressed by the regular expression $d\Gamma^*$. Thus, the subformula [call(FindNextFileA) $\wedge$ $d\Gamma^*$] expresses that *FindNextFileA* is called with d as parameter (d stores the information of the search handle). Therefore, φ_{sb} expresses then that there is a call to the API $FindFirstFileA$ with the return value d (the search handle), then, in all runs starting from that call, there will be either a call to the API function $GetLastError$ or a call to the function $FindFirstFileA$ or a call to the function $FindNextFileA$ in which d is used as a parameter.

To detect spyware, [13] used the following CARET formula:

$$\varphi'_{sb} = \bigvee_{d \in D} F^g(\text{call(FindFirstFileA)} \wedge X^a(eax = d) \wedge F^a(\text{call(FindNextFileA)} \wedge d\Gamma^*))$$

It can be seen that this CARET formula φ'_{sb} is not as precise as the BCARET formula φ_{sb}, as it does not deal with the case when the returned result of $FindFirstFileA$ is a folder or an error. Thus, this CARET formula φ'_{sb} may lead to false alarms that can be avoided using our BCARET formula φ_{sb}. BCARET can deal with it because BCARET is a branching-time temporal logic. For example, AF^a allows us to take into account all possible abstract-paths from a certain state in the computation tree. By using AF^a, φ_{sb} can deal with different returned values of $FindFirstFileA$ as presented above.

5 BCARET Model-Checking for Pushdown Systems

In this section, we consider "standard" BCARET model-checking for pushdown systems where an atomic proposition holds at a configuration c or not depends only on the control state of c, not on its stack.

5.1 Alternating Büchi Pushdown Systems (ABPDSs)

Definition 4. *An Alternating Büchi Pushdown System (ABPDS) is a tuple* $\mathcal{BP} = (P, \Gamma, \Delta, F)$*, where* P *is a set of control locations,* Γ *is the stack alphabet,* $F \subseteq P$ *is a set of accepting control locations and* Δ *is a transition function that maps each element of* $P \times \Gamma$ *with a positive boolean formula over* $P \times \Gamma^*$*.*

A configuration of $\mathcal{BP}$ is a pair $\langle p, \omega \rangle$, where $p \in P$ is the current control location and $\omega \in \Gamma^*$ is the current stack content. Without loss of generality, we suppose that the boolean formulas of ABPDSs are in disjunctive normal form $\bigvee_{j=1}^{n} \bigwedge_{i=1}^{m_j} \langle p_i^j, \omega_i^j \rangle$. Then, we can see Δ as a subset of $(P \times \Gamma) \times 2^{P \times \Gamma^*}$ by rewriting the rules of Δ in the form $\langle p, \gamma \rangle \rightarrow \bigvee_{j=1}^{n} \bigwedge_{i=1}^{m_j} \langle p_i^j, \omega_i^j \rangle$ as n rules of the form $\langle p, \gamma \rangle \rightarrow \{\langle p_1^j, \omega_1^j \rangle, ..., \langle p_{m_j}^j, \omega_{m_j}^j \rangle\}$, where $1 \leq j \leq n$. Let $\langle p, \gamma \rangle \rightarrow \{\langle p_1, \omega_1 \rangle, ..., \langle p_n, \omega_n \rangle\}$ be a rule of Δ, then, for every $\omega \in \Gamma^*$, the configuration $\langle p, \gamma\omega \rangle$(resp. $\{\langle p_1, \omega_1\omega \rangle, ..., \langle p_n, \omega_n\omega \rangle\}$) is an immediate predecessor (resp. successor) of $\{\langle p_1, \omega_1\omega \rangle, ..., \langle p_n, \omega_n\omega \rangle\}$ (resp. $\langle p, \gamma\omega \rangle$).

A run ρ of $\mathcal{BP}$ starting form an initial configuration $\langle p_0, \omega_0 \rangle$ is a tree whose root is labelled by $\langle p_0, \omega_0 \rangle$, and whose other nodes are labelled by elements in $P \times \Gamma^*$. If a node of ρ is labelled by a configuration $\langle p, \omega \rangle$ and has n children labelled by $\langle p_1, \omega_1 \rangle, ..., \langle p_n, \omega_n \rangle$ respectively, then, $\langle p, \omega \rangle$ must be a predecessor of $\{\langle p_1, \omega_1 \rangle, ..., \langle p_n, \omega_n \rangle\}$ in $\mathcal{BP}$. A path of a run ρ is an infinite sequence of configurations $c_0 c_1 c_2 ...$ s.t. c_0 is the root of ρ and c_{i+1} is one of the children of c_i for every $i \geq 0$. A path is accepting iff it visits infinitely often configurations with control locations in F. A run ρ is accepting iff every path of ρ is accepting. The language of $\mathcal{BP}$, $\mathcal{L}(\mathcal{BP})$, is the set of configurations c s.t. $\mathcal{BP}$ has an accepting run starting from c.

$\mathcal{BP}$ defines the reachability relation $\Rightarrow_{\mathcal{BP}_\varphi}$ as follows: (1) $c \Rightarrow_{\mathcal{BP}} \{c\}$ for every $c \in P \times \Gamma^*$, (2) $c \Rightarrow_{\mathcal{BP}} C$ if C is an immediate successor of c; (3) if $c \Rightarrow_{\mathcal{BP}} \{c_1, c_2, ..., c_n\}$ and $c_i \Rightarrow_{\mathcal{BP}} C_i$ for every $1 \leq i \leq n$, then $c \Rightarrow_{\mathcal{BP}} \bigcup_{i=1}^{n} C_i$. Given $c_0 \Rightarrow_{\mathcal{BP}} C'$, then, $\mathcal{BP}$ has an accepting run from c_0 iff $\mathcal{BP}$ has an accepting run from c' for every $c' \in C'$.

Theorem 1. *[15] Given an ABPDS $\mathcal{BP} = (P, \Gamma, \Delta, F)$, for every configuration $\langle p, \omega \rangle \in P \times \Gamma^*$, whether or not $\langle p, \omega \rangle \in \mathcal{L}(\mathcal{BP})$ can be decided in time $\mathcal{O}(|P|^2.|\Gamma|.(|\Delta|2^{5|P|} + 2^{|P|}|\omega|))$.*

5.2 From BCARET Model Checking of PDSs to the Membership Problem in ABPDSs

Let $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$ be a pushdown system with an initial configuration c_0. Given a set of atomic propositions AP, let φ be a BCARET formula. Let $f : AP \rightarrow 2^P$ be a function that associates each atomic proposition with a set of control states, and $\lambda_f : AP \rightarrow 2^{P \times \Gamma^*}$ be a labelling function s.t. for every $e \in AP$, $\lambda_f(e) = \{\langle p, \omega \rangle \mid p \in f(e), \omega \in \Gamma^*\}$. In this section, we propose an algorithm to check whether $c_0 \vDash_{\lambda_f} \varphi$. Intuitively, we construct an Alternating Büchi Pushdown System $\mathcal{BP}_\varphi$ which recognizes a configuration c iff $c \vDash_{\lambda_f} \varphi$. Then to check whether $c_0 \vDash_{\lambda_f} \varphi$, we will check if $c_0 \in \mathcal{L}(\mathcal{BP}_\varphi)$. The membership problem of an ABPDS can be solved effectively by Theorem 1.

Let $\mathcal{BP}_\varphi = (P', \Gamma', \Delta', F)$ be the ABPDS defined as follows:

- $P' = P \cup (P \times Cl(\varphi)) \cup \{p_\bot\}$
- $\Gamma' = \Gamma \cup (\Gamma \times Cl(\varphi)) \cup \{\gamma_\bot\}$
- $F = F_1 \cup F_2 \cup F_3$ where

- $F_1 = \{(\!|p, e|\!) \mid e \in Cl(\varphi), e \in AP \text{ and } p \in f(e)\}$
- $F_2 = \{(\!|p, \neg e|\!) \mid \neg e \in Cl(\varphi), e \in AP \text{ and } p \notin f(e)\}$
- $F_3 = \{P \times Cl_R(\varphi)\}$ where $Cl_R(\varphi)$ is the set of formulas of $Cl(\varphi)$ in the form $E[\varphi_1 R^b \varphi_2]$ or $A[\varphi_1 R^b \varphi_2]$ ($b \in \{g, a\}$)

The transition relation Δ' is the smallest set of transition rules defined as follows: $\Delta \subseteq \Delta'$ and for every $p \in P$, $\phi \in Cl(\varphi)$, $\gamma \in \Gamma$, $b \in \{g, a\}$ and $t \in \{call, ret, int\}$:

(α1) If $\phi = e$, $e \in AP$ and $p \in f(e)$, then, $\langle (\!|p, \phi|\!), \gamma\rangle \rightarrow \langle (\!|p, \phi|\!), \gamma\rangle \in \Delta'$
(α2) If $\phi = \neg e$, $e \in AP$ and $p \notin f(e)$, then, $\langle (\!|p, \phi|\!), \gamma\rangle \rightarrow \langle (\!|p, \phi|\!), \gamma\rangle \in \Delta'$
(α3) If $\phi = \phi_1 \wedge \phi_2$, then, $\langle (\!|p, \phi|\!), \gamma\rangle \rightarrow \langle (\!|p, \phi_1|\!), \gamma\rangle \wedge \langle (\!|p, \phi_2|\!), \gamma\rangle \in \Delta'$
(α4) If $\phi = \phi_1 \vee \phi_2$, then, $\langle (\!|p, \phi|\!), \gamma\rangle \rightarrow \langle (\!|p, \phi_1|\!), \gamma\rangle \vee \langle (\!|p, \phi_2|\!), \gamma\rangle \in \Delta'$
(α5) If $\phi = EX^g\phi_1$, then $(\!|p, \phi|\!), \gamma\rangle \rightarrow \bigvee_{\langle p,\gamma\rangle \xrightarrow{t} \langle q,\omega\rangle \in \Delta} \langle (\!|q, \phi_1|\!), \omega\rangle \in \Delta'$ where $t \in \{call, int, ret\}$
(α6)If $\phi = AX^g\phi_1$, then, $\langle (\!|p, \phi|\!), \gamma\rangle \rightarrow \bigwedge_{\langle p,\gamma\rangle \xrightarrow{t} \langle q,\omega\rangle \in \Delta} \langle (\!|q, \phi_1|\!), \omega\rangle \in \Delta'$
(α7) If $\phi = EX^a\phi_1$, then, $\langle (\!|p, \phi|\!), \gamma\rangle \rightarrow h_1 \vee h_2 \vee h_3 \in \Delta'$, where

- $h_1 = \bigvee_{\langle p,\gamma\rangle \xrightarrow{call} \langle q,\gamma_1\gamma_2\rangle \in \Delta} \langle q, \gamma_1 (\!|\gamma_2, \phi_1|\!)\rangle$
- $h_2 = \bigvee_{\langle p,\gamma\rangle \xrightarrow{int} \langle q,\omega\rangle \in \Delta} \langle (\!|q, \phi_1|\!), \omega\rangle$
- $h_3 = \bigvee_{\langle p,\gamma\rangle \xrightarrow{ret} \langle q,\epsilon\rangle \in \Delta} \langle p_\perp, \gamma_\perp\rangle$

(α8) If $\phi = AX^a\phi_1$, then, $\langle (\!|p, \phi|\!), \gamma\rangle \rightarrow h_1 \wedge h_2 \wedge h_3 \in \Delta'$, where

- $h_1 = \bigwedge_{\langle p,\gamma\rangle \xrightarrow{call} \langle q,\gamma_1\gamma_2\rangle \in \Delta} \langle q, \gamma_1 (\!|\gamma_2, \phi_1|\!)\rangle$
- $h_2 = \bigwedge_{\langle p,\gamma\rangle \xrightarrow{int} \langle q,\omega\rangle \in \Delta} \langle (\!|q, \phi_1|\!), \omega\rangle$
- $h_3 = \bigwedge_{\langle p,\gamma\rangle \xrightarrow{ret} \langle q,\epsilon\rangle \in \Delta} \langle p_\perp, \gamma_\perp\rangle$

(α9) If $\phi = E[\phi_1 U^g \phi_2]$, then, $\langle (\!|p, \phi|\!), \gamma\rangle \rightarrow \langle (\!|p, \phi_2|\!), \gamma\rangle \vee \bigvee_{\langle p,\gamma\rangle \xrightarrow{t} \langle q,\omega\rangle \in \Delta}$ $(\langle (\!|p, \phi_1|\!), \gamma\rangle \wedge \langle (\!|q, \phi|\!), \omega\rangle) \in \Delta'$
(α10) If $\phi = E[\phi_1 U^a \phi_2]$, then, $\langle (\!|p, \phi|\!), \gamma\rangle \rightarrow \langle (\!|p, \phi_2|\!), \gamma\rangle \vee h_1 \vee h_2 \vee h_3 \in \Delta'$, where

- $h_1 = \bigvee_{\langle p,\gamma\rangle \xrightarrow{call} \langle q,\gamma_1\gamma_2\rangle \in \Delta} \langle (\!|p, \phi_1|\!), \gamma\rangle \wedge \langle q, \gamma_1 (\!|\gamma_2, \phi|\!)\rangle$
- $h_2 = \bigvee_{\langle p,\gamma\rangle \xrightarrow{int} \langle q,\omega\rangle \in \Delta} \langle (\!|p, \phi_1|\!), \gamma\rangle \wedge \langle (\!|q, \phi|\!), \omega\rangle$
- $h_3 = \bigvee_{\langle p,\gamma\rangle \xrightarrow{ret} \langle q,\epsilon\rangle \in \Delta} \langle p_\perp, \gamma_\perp\rangle$

(α11) If $\phi = A[\phi_1 U^g \phi_2]$, then, $\langle (\!|p, \phi|\!), \gamma\rangle \rightarrow \langle (\!|p, \phi_2|\!), \gamma\rangle \vee \bigwedge_{\langle p,\gamma\rangle \xrightarrow{t} \langle q,\omega\rangle \in \Delta}$ $(\langle (\!|p, \phi_1|\!), \gamma\rangle \wedge \langle (\!|q, \phi|\!), \omega\rangle) \in \Delta'$
(α12) If $\phi = A[\phi_1 U^a \phi_2]$, then, $\langle (\!|p, \phi|\!), \gamma\rangle \rightarrow \langle (\!|p, \phi_2|\!), \gamma\rangle \vee (h_1 \wedge h_2 \wedge h_3) \in \Delta'$, where

- $h_1 = \bigwedge_{\langle p,\gamma\rangle \xrightarrow{call} \langle q,\gamma_1\gamma_2\rangle \in \Delta} \langle (\!|p, \phi_1|\!), \gamma\rangle \wedge \langle q, \gamma_1 (\!|\gamma_2, \phi|\!)\rangle$
- $h_2 = \bigwedge_{\langle p,\gamma\rangle \xrightarrow{int} \langle q,\omega\rangle \in \Delta} \langle (\!|p, \phi_1|\!), \gamma\rangle \wedge \langle (\!|q, \phi|\!), \omega\rangle$

- $h_3 = \bigwedge_{\langle p,\gamma\rangle \xrightarrow{ret} \langle q,\epsilon\rangle \in \Delta} \langle p_\bot, \gamma_\bot\rangle$

(α13) If $\phi = E[\phi_1 R^g \phi_2]$, then, we add to Δ' the rule: $\langle (\!|p, \phi|\!), \gamma\rangle \rightarrow (\langle (\!|p, \phi_2|\!), \gamma\rangle \wedge \langle (\!|p, \phi_1|\!), \gamma\rangle) \vee (\bigvee_{\langle p,\gamma\rangle \xrightarrow{t} \langle q,\omega\rangle \in \Delta} (\langle (\!|p, \phi_2|\!), \gamma\rangle \wedge \langle (\!|q, \phi|\!), \omega\rangle)$

(α14) If $\phi = A[\phi_1 R^g \phi_2]$, then, we add to Δ' the rule: $\langle (\!|p, \phi|\!), \gamma\rangle \rightarrow (\langle (\!|p, \phi_2|\!), \gamma\rangle \wedge \langle (\!|p, \phi_1|\!), \gamma\rangle) \vee (\bigwedge_{\langle p,\gamma\rangle \xrightarrow{t} \langle q,\omega\rangle \in \Delta} (\langle (\!|p, \phi_2|\!), \gamma\rangle \wedge \langle (\!|q, \phi|\!), \omega\rangle)$

(α15) If $\phi = E[\phi_1 R^a \phi_2]$: $\langle (\!|p, \phi|\!), \gamma\rangle \rightarrow (\langle (\!|p, \phi_2|\!), \gamma\rangle \wedge \langle (\!|p, \phi_1|\!), \gamma\rangle) \vee h_1 \vee h_2 \vee h_3 \in \Delta'$, where

- $h_1 = \bigvee_{\langle p,\gamma\rangle \xrightarrow{call} \langle q,\gamma_1\gamma_2\rangle \in \Delta} \langle (\!|p, \phi_2|\!), \gamma\rangle \wedge \langle q, \gamma_1 (\!|\gamma_2, \phi|\!)\rangle$
- $h_2 = \bigvee_{\langle p,\gamma\rangle \xrightarrow{int} \langle q,\omega\rangle \in \Delta} \langle (\!|p, \phi_2|\!), \gamma\rangle \wedge \langle (\!|q, \phi|\!), \omega\rangle$
- $h_3 = \bigvee_{\langle p,\gamma\rangle \xrightarrow{ret} \langle q,\epsilon\rangle \in \Delta} \langle p_\bot, \gamma_\bot\rangle$

(α16) If $\phi = A[\phi_1 R^a \phi_2]$, $\langle (\!|p, \phi|\!), \gamma\rangle \rightarrow (\langle (\!|p, \phi_2|\!), \gamma\rangle \wedge \langle (\!|p, \phi_1|\!), \gamma\rangle) \vee (h_1 \wedge h_2 \wedge h_3) \in \Delta'$, where

- $h_1 = \bigwedge_{\langle p,\gamma\rangle \xrightarrow{call} \langle q,\gamma_1\gamma_2\rangle \in \Delta} \langle (\!|p, \phi_2|\!), \gamma\rangle \wedge \langle q, \gamma_1 (\!|\gamma_2, \phi|\!)\rangle$
- $h_2 = \bigwedge_{\langle p,\gamma\rangle \xrightarrow{int} \langle q,\omega\rangle \in \Delta} \langle (\!|p, \phi_2|\!), \gamma\rangle \wedge \langle (\!|q, \phi|\!), \omega\rangle$
- $h_3 = \bigwedge_{\langle p,\gamma\rangle \xrightarrow{ret} \langle q,\epsilon\rangle \in \Delta} \langle p_\bot, \gamma_\bot\rangle$

(α17) for every $\langle p, \gamma\rangle \xrightarrow{ret} \langle q, \epsilon\rangle \in \Delta$:

- $\langle q, (\!|\gamma'', \phi_1|\!)\rangle \rightarrow \langle (\!|q, \phi_1|\!), \gamma''\rangle \in \Delta'$ for every $\gamma'' \in \Gamma$, $\phi_1 \in Cl(\varphi)$

(α18) $\langle p_\bot, \gamma_\bot\rangle \rightarrow \langle p_\bot, \gamma_\bot\rangle \in \Delta'$

Roughly speaking, the ABPDS $\mathcal{BP}_\varphi$ is a kind of product between $\mathcal{P}$ and the BCARET formula φ which ensures that $\mathcal{BP}_\varphi$ has an accepting run from $\langle (\!|p, \varphi|\!), \omega\rangle$ iff the configuration $\langle p, \omega\rangle$ satisfies φ. The form of the control locations of $\mathcal{BP}_\varphi$ is $(\!|p, \phi|\!)$ where $\phi \in Cl(\varphi)$. Let us explain the intuition behind our construction:

- If $\phi = e \in AP$, then, for every $\omega \in \Gamma^*$, $\langle p, \omega\rangle \vDash_{\lambda_f} \phi$ iff $p \in f(e)$. In other words, $\mathcal{BP}_\varphi$ should have an accepting run from $\langle (\!|p, e|\!), \omega\rangle$ iff $p \in f(e)$. This is ensured by the transition rules in (α1) which add a loop at $\langle (\!|p, e|\!), \omega\rangle$ where $p \in f(e)$ and the fact that $(\!|p, e|\!) \in F$.
- If $\phi = \neg e$ ($e \in AP$), then, for every $\omega \in \Gamma^*$, $\langle p, \omega\rangle \vDash_{\lambda_f} \phi$ iff $p \notin f(e)$. In other words, $\mathcal{BP}_\varphi$ should have an accepting run from $\langle (\!|p, \neg e|\!), \omega\rangle$ iff $p \notin f(e)$. This is ensured by the transition rules intransition rules in (α2) which add a loop at $\langle (\!|p, \neg e|\!), \omega\rangle$ where $p \notin f(e)$ and the fact that $(\!|p, \neg e|\!) \in F$.
- If $\phi = \phi_1 \wedge \phi_2$, then, for every $\omega \in \Gamma^*$, $\langle p, \omega\rangle \vDash_{\lambda_f} \phi$ iff ($\langle p, \omega\rangle \vDash_{\lambda_f} \phi_1$ and $\langle p, \omega\rangle \vDash_{\lambda_f} \phi_2$). This is ensured by the transition rules in (α3) stating that $\mathcal{BP}_\varphi$ has an accepting run from $\langle (\!|p, \phi_1 \wedge \phi_2|\!), \omega\rangle$ iff $\mathcal{BP}_\varphi$ has an accepting run from both $\langle (\!|p, \phi_1|\!), \omega\rangle$ and $\langle (\!|p, \phi_2|\!), \omega\rangle$. ($\alpha$4) is similar to ($\alpha$3).

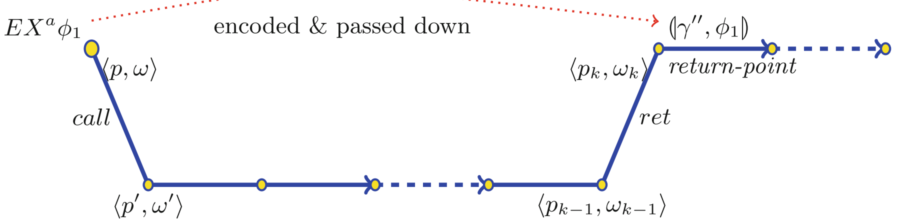


Fig. 2. $\langle p, \omega\rangle \Rightarrow_{\mathcal{P}} \langle p', \omega'\rangle$ corresponds to a call statement

– If $\phi = E[\phi_1 U^g \phi_2]$, then, for every $\omega \in \Gamma^*$, $\langle p, \omega\rangle \vDash_{\lambda_f} \phi$ iff $\langle p, \omega\rangle \vDash_{\lambda_f} \phi_2$ or $(\langle p, \omega\rangle \vDash_{\lambda_f} \phi_1$ and there exists an immediate successor $\langle p', \omega'\rangle$ of $\langle p, \omega\rangle$ s.t. $\langle p', \omega'\rangle \vDash_{\lambda_f} \phi)$. This is ensured by the transition rules in (α9) stating that $\mathcal{BP}_\varphi$ has an accepting run from $\langle (\!|p, E[\phi_1 U^g \phi_2]|\!), \omega\rangle$ iff $\mathcal{BP}_\varphi$ has an accepting run from $\langle (\!|p, \phi_2|\!), \omega\rangle$ or ($\mathcal{BP}_\varphi$ has an accepting run from both $\langle (\!|p, \phi_1|\!), \omega\rangle$ and $\langle (\!|p', \phi|\!), \omega'\rangle$ where $\langle p', \omega'\rangle$ is an immediate successor of $\langle p, \omega\rangle$). (α11) is similar to (α9).
– If $\phi = E[\phi_1 R^g \phi_2]$, then, for every $\omega \in \Gamma^*$, $\langle p, \omega\rangle \vDash_{\lambda_f} \phi$ iff $(\langle p, \omega\rangle \vDash_{\lambda_f} \phi_2$ and $\langle p, \omega\rangle \vDash_{\lambda_f} \phi_1)$ or $(\langle p, \omega\rangle \vDash_{\lambda_f} \phi_2$ and there exists an immediate successor $\langle p', \omega'\rangle$ of $\langle p, \omega\rangle$ s.t. $\langle p', \omega'\rangle \vDash_{\lambda_f} \phi)$. This is ensured by the transition rules in (α13) stating that $\mathcal{BP}_\varphi$ has an accepting run from $\langle (\!|p, E[\phi_1 R^g \phi_2]|\!), \omega\rangle$ iff $\mathcal{BP}_\varphi$ has an accepting run from both $\langle (\!|p, \phi_2|\!), \omega\rangle$ and $\langle (\!|p, \phi_1|\!), \omega\rangle$; or $\mathcal{BP}_\varphi$ has an accepting run from both $\langle (\!|p, \phi_2|\!), \omega\rangle$ and $(\!|p', \phi|\!), \omega'\rangle$ where $\langle p', \omega'\rangle$ is an immediate successor of $\langle p, \omega\rangle$. In addition, for R^g formulas, the *stop* condition is not required, i.e, for a formula $\phi_1 R^g \phi_2$ that is applied to a specific run, we don't require that ϕ_1 must eventually hold. To ensure that the runs on which ϕ_2 always holds are accepted, we add $(\!|p, \phi|\!)$ to the Büchi accepting condition F (via the subset F_3 of F). (α14) is similar to (α13).
– If $\phi = EX^g\phi_1$, then, for every $\omega \in \Gamma^*$, $\langle p, \omega\rangle \vDash_{\lambda_f} \phi$ iff there exists an immediate successor $\langle p', \omega'\rangle$ of $\langle p, \omega\rangle$ s.t. $\langle p', \omega'\rangle \vDash_{\lambda_f} \phi_1$. This is ensured by the transition rules in (α5) stating that $\mathcal{BP}_\varphi$ has an accepting run from $\langle (\!|p, EX^g\phi_1|\!), \omega\rangle$ iff there exists an immediate successor $\langle p', \omega'\rangle$ of $\langle p, \omega\rangle$ s.t. $\mathcal{BP}_\varphi$ has an accepting run from $\langle (\!|p', \phi_1|\!), \omega'\rangle$. ($\alpha$6) is similar to ($\alpha$5).
– If $\phi = EX^a\phi_1$, then, for every $\omega \in \Gamma^*$, $\langle p, \omega\rangle \vDash_{\lambda_f} \phi$ iff there exists an abstract-successor $\langle p_k, \omega_k\rangle$ of $\langle p, \omega\rangle$ s.t. $\langle p_k, \omega_k\rangle \vDash_{\lambda_f} \phi_1$ (A1). Let $\pi \in Traces(\langle p, \omega\rangle)$ be a run starting from $\langle p, \omega\rangle$ on which $\langle p_k, \omega_k\rangle$ is the abstract-successor of $\langle p, \omega\rangle$. Over π, let $\langle p', \omega'\rangle$ be the immediate successor of $\langle p, \omega\rangle$. In what follows, we explain how we can ensure (A1).

1. Firstly, we show that for every abstract-successor $\langle p_k, \omega_k\rangle \neq \bot$ of $\langle p, \omega\rangle$, $\langle (\!|p, EX^a\phi_1|\!), \omega\rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle (\!|p_k, \phi_1|\!), \omega_k\rangle$. There are two possibilities:

- If $\langle p, \omega\rangle \Rightarrow_{\mathcal{P}} \langle p', \omega'\rangle$ corresponds to a call statement. Let us consider Fig. 2 to explain this case. $\langle (\!|p, \phi|\!), \omega\rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle (\!|p_k, \phi_1|\!), \omega_k\rangle$ is ensured by rules corresponding to h_1 in (α7), the rules in $\Delta \subseteq \Delta'$ and the rules in (α17) as follows:

rules corresponding to h_1 in (α7) allow to record ϕ_1 in the return point of the call, rules in $\Delta \subseteq \Delta'$ allow to mimic the run of the PDS $\mathcal{P}$ and rules in (α17) allow to extract and put back ϕ_1 when the return-point is reached. In what follows, we show in more details how this works: Let $\langle p, \gamma\rangle \xrightarrow{call} \langle p', \gamma'\gamma''\rangle$ be the rule associated with the transition $\langle p, \omega\rangle \Rightarrow_{\mathcal{P}} \langle p', \omega'\rangle$, then we have $\omega = \gamma\omega''$ and $\omega' = \gamma'\gamma''\omega''$. Let $\langle p_{k-1}, \omega_{k-1}\rangle \Rightarrow_{\mathcal{P}} \langle p_k, \omega_k\rangle$ be the transition that corresponds to the *ret* statement of this call on π. Let then $\langle p_{k-1}, \beta\rangle \xrightarrow{ret} \langle p_k, \epsilon\rangle \in \Delta$ be the corresponding return rule. Then, we have necessarily $\omega_{k-1} = \beta\gamma''\omega''$, since as explained in Sect. 2, γ'' is the return address of the call. After applying this rule, $\omega_k = \gamma''\omega''$. In other words, γ'' will be the topmost stack symbol at the corresponding return point of the call. So, in order to ensure that $\langle (\!|p, \phi|\!), \omega\rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle (\!|p_k, \phi_1|\!), \omega_k\rangle$, we proceed as follows: At the call $\langle p, \gamma\rangle \xrightarrow{call} \langle p', \gamma'\gamma''\rangle$, we encode the formula ϕ_1 into γ'' by the rule corresponding to h_1 in (α7) stating that $\langle (\!|p, EX^a\phi_1|\!), \gamma\rangle \rightarrow \langle p', \gamma'(\!|\gamma'', \phi_1|\!)\rangle \in \Delta'$. This allows to record ϕ_1 in the corresponding return point of the stack. After that, the rules in $\Delta \subseteq \Delta'$ allow $\mathcal{BP}_\varphi$ to mimic the run π of $\mathcal{P}$ from $\langle p', \omega'\rangle$ till the corresponding return-point of this call, where $(\!|\gamma'', \phi_1|\!)$ is the topmost stack symbol. More specifically, the following sequence of $\mathcal{P}$: $\langle p', \gamma'\gamma''\omega''\rangle \overset{*}{\Rightarrow}_{\mathcal{P}} \langle p_{k-1}, \beta\gamma''\omega''\rangle \overset{*}{\Rightarrow}_{\mathcal{P}} \langle p_k, \gamma''\omega''\rangle$ will be mimicked by the following sequence of $\mathcal{BP}_\varphi$: $\langle (\!|p', \gamma'(\!|\gamma'', \phi_1|\!)\omega''\rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle p_{k-1}, \beta(\!|\gamma'', \phi_1|\!)\omega''\rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle p_k, (\!|\gamma'', \phi_1|\!)\omega''\rangle$ using the rules of Δ. At the return-point, we extract ϕ_1 from the stack and encode it into p_k by adding the transition rules in (α17) $\langle p_k, (\!|\gamma'', \phi_1|\!)\rangle \rightarrow \langle (\!|p_k, \phi_1|\!), \gamma''\rangle$. Therefore, we obtain that $\langle (\!|p, \phi|\!), \omega\rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle (\!|p_k, \phi_1|\!), \omega_k\rangle$. The property holds for this case.

- If $\langle p, \omega\rangle \Rightarrow_{\mathcal{P}} \langle p', \omega'\rangle$ corresponds to a simple statement. Then, the abstract successor of $\langle p, \omega\rangle$ is its immediate successor $\langle p', \omega'\rangle$. Thus, we get that $\langle p_k, \omega_k\rangle = \langle p', \omega'\rangle$. From the transition rules corresponding to h_2 in (α7), we get that $\langle (\!|p, EX^a\phi_1|\!), \omega\rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle (\!|p', \phi_1|\!), \omega'\rangle$. Therefore, $\langle (\!|p, EX^a\phi_1|\!), \omega\rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle (\!|p_k, \phi_1|\!), \omega_k\rangle$. The property holds for this case.

2. Now, let us consider the case where $\langle p_k, \omega_k\rangle$, the abstract successor of $\langle p, \omega\rangle$, is $\perp$. This case occurs when $\langle p, \omega\rangle \Rightarrow_{\mathcal{P}} \langle p', \omega'\rangle$ corresponds to a return statement. Then, one abstract successor of $\langle p, \omega\rangle$ is $\perp$. Note that $\perp$ does not satisfy any formula, i.e., $\perp$ does not satisfy ϕ_1. Therefore, from $\langle (\!|p, EX^a\phi_1|\!), \omega\rangle$, we need to ensure that the path of $\mathcal{BP}_\varphi$ reflecting the possibility in (A1) that $\langle p_k, \omega_k\rangle \vDash_{\lambda_f} \phi_1$ is not accepted. To do this, we exploit additional trap configurations. We use $p_\perp$ and $\gamma_\perp$ as trap control location and trap stack symbol to obtain these trap configurations. To be more specific, let $\langle p, \gamma\rangle \xrightarrow{ret} \langle p', \epsilon\rangle$ be the rule associated with the transition $\langle p, \omega\rangle \Rightarrow_{\mathcal{P}} \langle p', \omega'\rangle$, then we have $\omega = \gamma\omega''$ and $\omega' = \omega''$. We add the transition rule corresponding to h_3 in (α7) to allow $\langle (\!|p, EX^a\phi_1|\!), \omega\rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle p_\perp, \gamma_\perp\omega''\rangle$. Since a run of $\mathcal{BP}_\varphi$ includes only infinite paths, we equip these trap configurations with self-loops by the transition rules in (α18), i.e., $\langle p_\perp, \gamma_\perp\omega''\rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle p_\perp, \gamma_\perp\omega''\rangle$. As a result, we obtain a corresponding path in $\mathcal{BP}_\varphi$: $\langle (\!|p, EX^a\phi_1|\!), \omega\rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle p_\perp, \gamma_\perp\omega''\rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle p_\perp, \gamma_\perp\omega''\rangle$. Note that this path is not accepted by $\mathcal{BP}_\varphi$ because $p_\perp \notin F$.

In summary, for every abstract-successor $\langle p_k, \omega_k \rangle$ of $\langle p, \omega \rangle$, if $\langle p_k, \omega_k \rangle \neq \bot$, then, $\langle (\!| p, EX^a\phi_1 |\!), \omega \rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle (\!| p_k, \phi_1 |\!), \omega_k \rangle$; otherwise $\langle (\!| p, EX^a\phi_1 |\!), \omega \rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle p_\bot, \gamma_\bot \omega'' \rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle p_\bot, \gamma_\bot \omega'' \rangle$ which is not accepted by $\mathcal{BP}_\varphi$. Therefore, (A1) is ensured by the transition rules in (α7) stating that $\mathcal{BP}_\varphi$ has an accepting run from $\langle (\!| p, EX^a\phi_1 |\!), \omega \rangle$ iff there exists an abstract successor $\langle p_k, \omega_k \rangle$ of $\langle p, \omega \rangle$ s.t. $\mathcal{BP}_\varphi$ has an accepting run from $\langle (\!| p_k, \phi_1 |\!), \omega_k \rangle$.

- If $\phi = AX^a\phi_1$: this case is ensured by the transition rules in (α8) together with (α17) and $\Delta \subseteq \Delta'$. The intuition of (α8) is similar to that of (α7).
- If $\phi = E[\phi_1 U^a \phi_2]$, then, for every $\omega \in \Gamma^*$, $\langle p, \omega \rangle \vDash_{\lambda_f} \phi$ iff $\langle p, \omega \rangle \vDash_{\lambda_f} \phi_2$ or $(\langle p, \omega \rangle \vDash_{\lambda_f} \phi_1$ and there exists an abstract successor $\langle p_k, \omega_k \rangle$ of $\langle p, \omega \rangle$ s.t. $\langle p_k, \omega_k \rangle \vDash_{\lambda_f} \phi)$ (A2). Let $\pi \in Traces(\langle p, \omega \rangle)$ be a run starting from $\langle p, \omega \rangle$ on which $\langle p_k, \omega_k \rangle$ is the abstract-successor of $\langle p, \omega \rangle$. Over π, let $\langle p', \omega' \rangle$ be the immediate successor of $\langle p, \omega \rangle$.

1. Firstly, we show that for every abstract-successor $\langle p_k, \omega_k \rangle \neq \bot$ of $\langle p, \omega \rangle, \langle (\!| p, \phi |\!), \omega \rangle \Rightarrow_{\mathcal{BP}_\varphi} \{\langle (\!| p, \phi_1 |\!), \omega \rangle, \langle (\!| p_k, \phi |\!), \omega_k \rangle\}$. There are two possibilities:

- If $\langle p, \omega \rangle \Rightarrow_{\mathcal{P}} \langle p', \omega' \rangle$ corresponds to a call statement. From the rules corresponding to h_1 in (α10), we get that $\langle (\!| p, \phi |\!), \omega \rangle \Rightarrow_{\mathcal{BP}_\varphi} \{\langle (\!| p, \phi_1 |\!), \omega \rangle, \langle p', \omega' \rangle\}$ where $\langle p', \omega' \rangle$ is the immediate successor of $\langle p, \omega \rangle$. Thus, to ensure that $\langle (\!| p, \phi |\!), \omega \rangle \Rightarrow_{\mathcal{BP}_\varphi} \{\langle (\!| p, \phi_1 |\!), \omega \rangle, \langle (\!| p_k, \phi |\!), \omega_k \rangle\}$, we only need to ensure that $\langle p', \omega' \rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle (\!| p_k, \phi |\!), \omega_k \rangle$. As for the case $\phi = EX^a\phi_1$, $\langle p', \omega' \rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle (\!| p_k, \phi |\!), \omega_k \rangle$ is ensured by the rules in $\Delta \subseteq \Delta'$ and the rules in (α17): rules in $\Delta \subseteq \Delta'$ allow to mimic the run of the PDS $\mathcal{P}$ before the return and rules in (α17) allow to extract and put back ϕ_1 when the return-point is reached.
- If $\langle p, \omega \rangle \Rightarrow_{\mathcal{P}} \langle p', \omega' \rangle$ corresponds to a simple statement. Then, the abstract successor of $\langle p, \omega \rangle$ is its immediate successor $\langle p', \omega' \rangle$. Thus, we get that $\langle p_k, \omega_k \rangle = \langle p', \omega' \rangle$. From the transition rules corresponding to h_2 in (α10), we get that $\langle (\!| p, E[\phi_1 U^a \phi_2] |\!), \omega \rangle \Rightarrow_{\mathcal{BP}_\varphi} \{\langle (\!| p, \phi_1 |\!), \omega \rangle, \langle (\!| p', \phi |\!), \omega' \rangle\}$. Therefore, $\langle (\!| p, E[\phi_1 U^a \phi_2] |\!), \omega \rangle \Rightarrow_{\mathcal{BP}_\varphi} \{\langle (\!| p, \phi_1 |\!), \omega \rangle, \langle (\!| p_k, \phi |\!), \omega_k \rangle\}$. In other words, $\mathcal{BP}_\varphi$ has an accepting run from both $\langle (\!| p, \phi_1 |\!), \omega \rangle$ and $\langle (\!| p_k, \phi |\!), \omega_k \rangle$ where $\langle p_k, \omega_k \rangle$ is an abstract successor of $\langle p, \omega \rangle$. The property holds for this case.

2. Now, let us consider the case where $\langle p_k, \omega_k \rangle = \bot$. As explained previously, this case occurs when $\langle p, \omega \rangle \Rightarrow_{\mathcal{P}} \langle p', \omega' \rangle$ corresponds to a return statement. Then, the abstract successor of $\langle p, \omega \rangle$ is $\bot$. Note that $\bot$ does not satisfy any formula, i.e., $\bot$ does not satisfy ϕ. Therefore, from $\langle (\!| p, E[\phi_1 U^a \phi_2] |\!), \omega \rangle$, we need to ensure that the path reflecting the possibility in (A2) that $(\langle p, \omega \rangle \vDash_{\lambda_f} \phi_1$ and $\langle p_k, \omega_k \rangle \vDash_{\lambda_f} \phi)$ is not accepted by $\mathcal{BP}_\varphi$. This is ensured as for the case $\phi = EX^a\phi_1$ by the transition rules corresponding to h_3 in (α10).

In summary, for every abstract-successor $\langle p_k, \omega_k \rangle$ of $\langle p, \omega \rangle$, if $\langle p_k, \omega_k \rangle \neq \bot$, then, $\langle (\!| p, E[\phi_1 U^a \phi_2] |\!), \omega \rangle \Rightarrow_{\mathcal{BP}_\varphi} \{\langle (\!| p, \phi_1 |\!), \omega \rangle, \langle (\!| p_k, E[\phi_1 U^a \phi_2] |\!), \omega_k \rangle\}$; otherwise $\langle (\!| p, E[\phi_1 U^a \phi_2] |\!), \omega \rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle p_\bot, \gamma_\bot \omega'' \rangle \Rightarrow_{\mathcal{BP}_\varphi} \langle p_\bot, \gamma_\bot \omega'' \rangle$ which is not accepted by $\mathcal{BP}_\varphi$. Therefore, (A2) is ensured by the transition rules in (α10) stating that

$\mathcal{BP}_\varphi$ has an accepting run from $\langle (\!|p, E[\phi_1 U^a \phi_2]|\!), \omega\rangle$ iff $\mathcal{BP}_\varphi$ has an accepting run from $\langle (\!|p, \phi_2|\!), \omega\rangle$; or $\mathcal{BP}_\varphi$ has an accepting run from both $\langle (\!|p, \phi_1|\!), \omega\rangle$ and $\langle (\!|p_k, E[\phi_1 U^a \phi_2]|\!), \omega_k\rangle$ where $\langle p_k, \omega_k\rangle$ is an abstract successor of $\langle p, \omega\rangle$.

– The intuition behind the rules corresponding to the cases $\phi = A[\phi_1 U^a \phi_2]$, $\phi = E[\phi_1 R^a \phi_2]$, $\phi = A[\phi_1 R^a \phi_2]$ are similar to the previous cases.

The Büchi Accepting Condition. The elements of the Büchi accepting condition set F of $\mathcal{BP}_\varphi$ ensure the liveness requirements of until-formulas on infinite global paths, infinite abstract paths as well as on finite abstract paths.

– With regards to infinite global paths, the fact that the liveness requirement ϕ_2 in $E[\phi_1 U^g \phi_2]$ is eventually satisfied in $\mathcal{P}$ is ensured by the fact that $(\!|p, E[\phi_1 U^g \phi_2]|\!)$ doesn't belong to F. Note that $\langle p, \omega\rangle \vDash_{\lambda_f} E[\phi_1 U^g \phi_2]$ iff $\langle p, \omega\rangle \vDash_{\lambda_f} \phi_2$ or there exists a global-successor $\langle p', \omega'\rangle$ s.t. $(\langle p, \omega\rangle \vDash_{\lambda_f} \phi_1$ and $\langle p', \omega'\rangle \vDash_{\lambda_f} E[\phi_1 U^g \phi_2])$. Because ϕ_2 should hold eventually, to avoid the case where a run of $\mathcal{BP}_\varphi$ always carries $E[\phi_1 U^g \phi_2]$ and never reaches ϕ_2, we don't set $(\!|p, E[\phi_1 U^g \phi_2]|\!)$ as an element of the Büchi accepting condition set. This guarantees that the accepting run of $\mathcal{BP}_\varphi$ must visit some control locations in $(\!|p, \phi_2|\!)$ which ensures that ϕ_2 will eventually hold. The liveness requirements of $A[\phi_1 U^g \phi_2]$ are ensured as for the case of $E[\phi_1 U^g \phi_2]$.
– With regards to infinite abstract paths, the fact that the liveness requirement ϕ_2 in $E[\phi_1 U^a \phi_2]$ is eventually satisfied in $\mathcal{P}$ is ensured by the fact that $(\!|p, E[\phi_1 U^a \phi_2]|\!)$ doesn't belong to F. The intuition behind this case is similar to the intuition of $E[\phi_1 U^g \phi_2]$. The liveness requirements of $A[\phi_1 U^a \phi_2]$ are ensured as for the case of $E[\phi_1 U^a \phi_2]$.

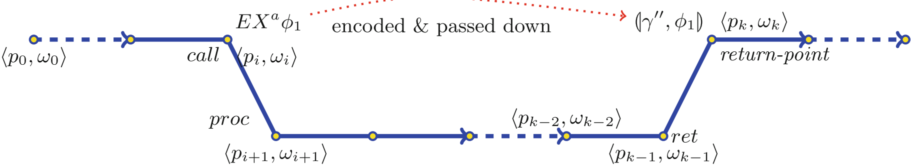


Fig. 3. $\langle p_i, \omega_i\rangle$ finally reach its corresponding return-point

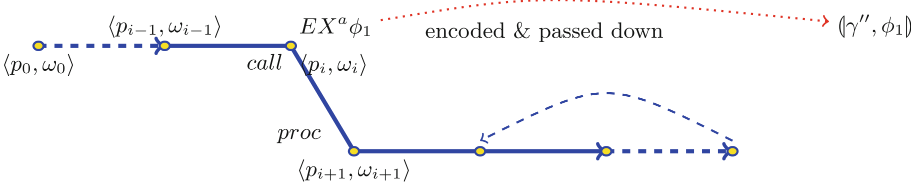


Fig. 4. $\langle p_i, \omega_i\rangle$ never reach its corresponding return-point

- With regards to finite abstract paths $\langle p_0, \omega_0\rangle\langle p_1, \omega_1\rangle...\langle p_m, \omega_m\rangle$ where $\langle p_m, \omega_m\rangle \Rightarrow_{\mathcal{P}} \langle p_{m+1}, \omega_{m+1}\rangle$ corresponds to a return statement, the fact that the liveness requirement ϕ_2 in $E[\phi_1 U^a \phi_2]$ is eventually satisfied in $\mathcal{P}$ is ensured by the fact that $p_\bot$ doesn't belong to F. Look at Fig. 3 for an illustration. In this figure, for every $i+1 \leq u \leq k-1$, the abstract path starting from $\langle p_u, \omega_u\rangle$ is finite because the abstract successor of $\langle p_{k-1}, \omega_{k-1}\rangle$ is $\bot$ since $\langle p_{k-1}, \omega_{k-1}\rangle \Rightarrow_{\mathcal{P}} \langle p_k, \omega_k\rangle$ corresponds to a return statement. Suppose that we want to check whether $\langle p_{k-1}, \omega_{k-1}\rangle \vDash_{\lambda_f} E[\phi_1 U^a \phi_2]$, then, we get that $\langle p_{k-1}, \omega_{k-1}\rangle \vDash_{\lambda_f} E[\phi_1 U^a \phi_2]$ iff $\langle p_{k-1}, \omega_{k-1}\rangle \vDash_{\lambda_f} \phi_2$ or there exists an abstract-successor $\langle p', \omega'\rangle$ s.t. $(\langle p_{k-1}, \omega_{k-1}\rangle \vDash_{\lambda_f} \phi_1$ and $\langle p', \omega'\rangle \vDash_{\lambda_f} E[\phi_1 U^a \phi_2])$. Since ϕ_2 should eventually hold, ϕ_2 should hold at $\langle p_{k-1}, \omega_{k-1}\rangle$ because the abstract-successor of $\langle p_{k-1}, \omega_{k-1}\rangle$ on this abstract-path is $\bot$. To ensure this, we move $\langle p_{k-1}, \omega_{k-1}\rangle$ to the trap configuration $\langle p_\bot, \gamma_\bot\rangle$ and add a loop here by the transition rule (α18). In addition, we don't set $p_\bot$ as an element of the Büchi accepting condition set, which means that $\langle p_{k-1}, \omega_{k-1}\rangle \vDash_{\lambda_f} E[\phi_1 U^a \phi_2]$ iff $\langle p_{k-1}, \omega_{k-1}\rangle \vDash_{\lambda_f} \phi_2$ by the transition rules in (α10). This ensures the liveness requirement ϕ_2 in $E[\phi_1 U^a \phi_2]$ is eventually satisfied.
- With regards to finite abstract paths $\langle p_0, \omega_0\rangle\langle p_1, \omega_1\rangle...\langle p_m, \omega_m\rangle$ where $\langle p_m, \omega_m\rangle \Rightarrow_{\mathcal{P}} \langle p_{m+1}, \omega_{m+1}\rangle$ corresponds to a call statement but this call never reaches its corresponding return-point, the fact that the liveness requirement ϕ_2 in $E[\phi_1 U^a \phi_2]$ is eventually satisfied in $\mathcal{P}$ is ensured by the fact that $p \notin F$. Look at Fig. 4 where the procedure *proc* never terminates. In this figure, for every $0 \leq u \leq i$, the abstract path starting from $\langle p_u, \omega_u\rangle$ is finite. Suppose that we want to check whether $\langle p_i, \omega_i\rangle \vDash_{\lambda_f} E[\phi_1 U^a \phi_2]$, then, we get that $\langle p_i, \omega_i\rangle \vDash_{\lambda_f} E[\phi_1 U^a \phi_2]$ iff $\langle p_i, \omega_i\rangle \vDash_{\lambda_f} \phi_2$ or there exists an abstract-successor $\langle p', \omega'\rangle$ s.t. $(\langle p_i, \omega_i\rangle \vDash_{\lambda_f} \phi_1$ and $\langle p', \omega'\rangle \vDash_{\lambda_f} E[\phi_1 U^a \phi_2])$. Since ϕ_2 should eventually hold, ϕ_2 should hold at $\langle p_i, \omega_i\rangle$ because the abstract-successor of $\langle p_i, \omega_i\rangle$ on this abstract-path is $\bot$. As explained above, at $\langle p_i, \omega_i\rangle$, we will encode the formula $E[\phi_1 U^a \phi_2]$ into the stack and mimic the run of $\mathcal{P}$ on $\mathcal{BP}_\varphi$ until it reaches the corresponding return-point of the call. In other words, if the call is never reached, the run of $\mathcal{BP}_\varphi$ will infinitely visit the control locations of $\mathcal{P}$. To ensure this path unaccepted, we don't set $p \in P$ as an element of the Büchi accepting condition set, which means that $\langle p_i, \omega_i\rangle \vDash_{\lambda_f} E[\phi_1 U^a \phi_2]$ iff $\langle p_i, \omega_i\rangle \vDash_{\lambda_f} \phi_2$ by the transition rules in (α10). This ensures the liveness requirement ϕ_2 in $E[\phi_1 U^a \phi_2]$ is eventually satisfied.

Thus, we can show that:

Theorem 2. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$, a set of atomic propositions AP, a labelling function $f : AP \to 2^P$ and a BCARET formula φ, we can compute an ABPDS $\mathcal{BP}_\varphi$ such that for every configuration $\langle p, \omega\rangle$, $\langle p, \omega\rangle \vDash_{\lambda_f} \varphi$ iff $\mathcal{BP}_\varphi$ has an accepting run from the configuration $\langle (\!| p, \varphi |\!), \omega\rangle$.*

The number of control locations of $\mathcal{BP}_\varphi$ is at most $\mathcal{O}(|P||\varphi|)$, the number of stack symbols is at most $\mathcal{O}(|\Gamma||\varphi|)$ and the number of transitions is at most $\mathcal{O}(|P||\Gamma||\Delta||\varphi|)$. Therefore, we get from Theorems 1 and 2:

Theorem 3. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$, a set of atomic propositions AP, a labelling function $f : AP \to 2^P$ and a BCARET formula φ, for every configuration $\langle p, \omega\rangle \in P \times \Gamma^*$, whether or not $\langle p, \omega\rangle$ satisfies φ can be solved in time $\mathcal{O}(|P|^2|\varphi|^3.|\Gamma|(|P||\Gamma||\Delta|.|\varphi|.2^{5|P||\varphi|} + 2^{|P||\varphi|}.|\omega|))$.*

6 BCARET Model-Checking for PDSs with Regular Valuations

Up to now, we have considered the *standard* model-checking problem for BCARET, where the validity of an atomic proposition depends only on the control state, not on the stack. In this section, we go further and consider model-checking with regular valuations where the set of configurations in which an atomic proposition holds is a regular set of configurations (see Sect. 3 for a formal definition of regular valuations).

6.1 From BCARET Model Checking of PDSs with Regular Valuations to the Membership Problem in ABPDSs

Given a pushdown system $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$, and a set of atomic propositions AP, let φ be a BCARET formula over AP, $\lambda : AP \to 2^{P\times\Gamma^*}$ be a labelling function s.t. for every $e \in AP$, $\lambda(e)$ is a regular set of configurations. Given a configuration c_0, we propose in this section an algorithm to check whether $c_0 \vDash_\lambda \varphi$. Intuitively, we compute an ABPDS $\mathcal{BP}'_\varphi$ s.t. $\mathcal{BP}'_\varphi$ recognizes a configuration c of $\mathcal{P}$ iff $c \vDash_\lambda \varphi$. Then, to check if c_0 satisfies φ, we will check whether $\mathcal{BP}'_\varphi$ recognizes c_0.

For every $e \in AP$, since $\lambda(e)$ is a regular set of configurations, let $M_e = (Q_e, \Gamma, \delta_e, I_e, F_e)$ be a multi-automaton s.t. $L(M_e) = \lambda(e)$, $M_{\neg e} = (Q_{\neg e}, \Gamma, \delta_{\neg e}, I_{\neg e}, F_{\neg e})$ be a multi-automaton s.t. $L(M_{\neg e}) = P \times \Gamma^* \setminus \lambda(e)$, which means $M_{\neg e}$ will recognize the complement of $\lambda(e)$ that is the set of configurations in which e doesn't hold. Note that for every $e \in AP$, the initial states of M_e and $M_{\neg e}$ are the control locations $p \in \mathcal{P}$. Thus, to distinguish between the initial states of these two automata, we will denote the initial state corresponding to the control location p in M_e (resp. $M_{\neg e}$) by p_e (resp. $p_{\neg e}$). Let $AP^+(\varphi) = \{e \in AP \mid e \in Cl(\varphi)\}$ and $AP^-(\varphi) = \{e \in AP \mid \neg e \in Cl(\varphi)\}$.

Let $\mathcal{BP}'_\varphi = (P'', \Gamma'', \Delta'', F')$ be the ABPDS defined as follows:

- $P'' = P \cup P \times Cl(\varphi) \cup \{p_\bot\} \cup \bigcup_{e\in AP^+(\varphi)} Q_e \cup \bigcup_{e\in AP^-(\varphi)} Q_{\neg e}$
- $\Gamma'' = \Gamma \cup (\Gamma \times Cl(\varphi)) \cup \{\gamma_\bot\}$
- $F' = F_1 \cup F_2 \cup F_3$ where
  - $F_1 = \bigcup_{e\in AP^+(\varphi)} F_e$
  - $F_2 = \bigcup_{e\in AP^-(\varphi)} F_{\neg e}$
  - $F_3 = \{P \times Cl_R(\varphi)\}$ where $Cl_R(\varphi)$ is the set of formulas of $Cl(\varphi)$ in the form $E[\varphi_1 R^b \varphi_2]$ or $A[\varphi_1 R^b \varphi_2]$ $(b \in \{g, a\})$

The transition relation Δ'' is the smallest set of transition rules defined as follows: $\Delta \subseteq \Delta''$, $\Delta'_0 \subseteq \Delta''$ where Δ'_0 is the transitions of Δ' that are created by the rules from $(\alpha 3)$ to $(\alpha 18)$ and such that:

($\beta 1$) for every $p \in P$, $e \in AP^{+}(\varphi)$, $\gamma \in \Gamma$: $\langle (\!|p, e|\!), \gamma \rangle \rightarrow \langle p_e, \gamma \rangle \in \Delta''$
($\beta 2$) for every $p \in P$, $e \in AP^{-}(\varphi)$, $\gamma \in \Gamma$: $\langle (\!|p, \neg e|\!), \gamma \rangle \rightarrow \langle p_{\neg e}, \gamma \rangle \in \Delta''$
($\beta 3$) for very $(q_1, \gamma, q_2) \in (\bigcup_{e \in AP^{+}(\varphi)} \delta_e) \cup (\bigcup_{e \in AP^{-}(\varphi)} \delta_{\neg e})$: $\langle q_1, \gamma \rangle \rightarrow \langle q_2, \epsilon \rangle \in \Delta''$
($\beta 4$) for very $q \in (\bigcup_{e \in AP^{+}(\varphi)} F_e) \cup (\bigcup_{e \in AP^{-}(\varphi)} F_{\neg e})$: $\langle q, \sharp \rangle \rightarrow \langle q, \sharp \rangle \in \Delta''$

Intuitively, we compute the ABPDS $\mathcal{BP}'_\varphi$ such that $\mathcal{BP}'_\varphi$ has an accepting run from $\langle (\!|p, \phi|\!), \omega \rangle$ iff the configuration $\langle p, \omega \rangle$ satisfies ϕ according to the regular labellings M_e for every $e \in AP$. The only difference with the previous case of standard valuations, where an atomic proposition holds at a configuration depends only on the control location of that configuration, not on its stack, comes from the interpretation of the atomic proposition e. This is why Δ'' contains Δ and Δ'_0 (which are the transitions of $\mathcal{BP}_\varphi$ that don't consider the atomic propositions). Here the rules $(\beta_1) - (\beta_4)$ deal with the cases e, $\neg e$ ($e \in AP$). Given $p \in P$, $\phi = e \in AP$, $\omega \in \Gamma^*$, we get that the ABPDS $\mathcal{BP}'_\varphi$ should accept $\langle (\!|p, e|\!), \omega \rangle$ iff $\langle p, \omega \rangle \in L(M_e)$. To check whether $\langle p, \omega \rangle \in L(M_e)$, we let $\mathcal{BP}'_\varphi$ go to state p_e, the initial state corresponding to p in M_e by adding rules in ($\beta 1$); and then, from this state, we will check whether ω is accepted by M_e. This is ensured by the transition rules in ($\beta 3$) and ($\beta 4$). ($\beta 3$) lets $\mathcal{BP}'_\varphi$ mimic a run of M_e on ω, i.e., if $\mathcal{BP}'_\varphi$ is in a state q_1 with γ on the top of the stack, and if (q_1, γ, q_2) is a transition rule in M_e, then, $\mathcal{BP}'_\varphi$ will move to state q_2 and pop γ from its stack. Note that popping γ allows us to check the rest of the word. In M_e, a configuration is accepted if the run with the word ω reaches the final state in F_e; i.e., if $\mathcal{BP}'_\varphi$ reaches a state $q \in F_e$ with an empty stack, i.e., with a stack containing the bottom stack symbol $\sharp$. Thus, we add F_e as a set of accepting control locations in $\mathcal{BP}'_\varphi$. Since $\mathcal{BP}'_\varphi$ only recognizes infinite paths, ($\beta 4$) adds a loop on every configuration $\langle q, \sharp \rangle$ where $q \in F_e$. The intuition behind the transition rules in ($\beta 2$) is similar to that of ($\beta 1$). They correspond to the case where $\phi = \neg e$.

Theorem 4. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$, a set of atomic propositions AP, a regular labelling function $\lambda : AP \rightarrow 2^{P \times \Gamma^*}$ and a BCARET formula φ, we can compute an ABPDS $\mathcal{BP}'_\varphi$ such that for every configuration $\langle p, \omega \rangle$, $\langle p, \omega \rangle \vDash_\lambda \varphi$ iff $\mathcal{BP}'_\varphi$ has an accepting run from the configuration $\langle (\!|p, \varphi|\!), \omega \rangle$*

The number of control locations of $\mathcal{BP}'_\varphi$ is at most $\mathcal{O}(|P||\varphi| + k)$ where $k = \sum_{e \in AP^{+}(\varphi)} |Q_e| + \sum_{e \in AP^{-}(\varphi)} |Q_{\neg e}|$, the number of stack symbols is at most $\mathcal{O}(|\Gamma||\varphi|)$ and the number of transitions is at most $\mathcal{O}(|P||\Gamma||\Delta||\varphi| + d)$ where $d = \sum_{e \in AP^{+}(\varphi)} |\delta_e| + \sum_{e \in AP^{-}(\varphi)} |\delta_{\neg e}|$. Therefore, we get from Theorems 1 and 4.

Theorem 5. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$, a set of atomic propositions AP, a regular labelling function $\lambda : AP \rightarrow 2^{P \times \Gamma^*}$ and a BCARET formula φ, for every configuration $\langle p, \omega \rangle \in P \times \Gamma^*$, whether or not $\langle p, \omega \rangle$ satisfies φ can be solved in time $\mathcal{O}((|P||\varphi| + k)^2.|\Gamma||\varphi|((|P||\Gamma||\Delta||\varphi| + d).2^{5(|P||\varphi|+k)} + 2^{|P||\varphi|+k}.|\omega|))$.*

7 Conclusion

In this paper, we introduce the Branching temporal logic of CAlls and RETurns BCARET and show how it can be used to describe malicious behaviors that CARET and other specification formalisms cannot. We present an algorithm for "standard" BCARET model checking for PDSs where whether a configuration of a PDS satisfies an atomic proposition or not depends only on the control location of that configuration. Moreover, we consider BCARET model-checking for PDSs with regular valuations where the set of configurations on which an atomic proposition holds is a regular language. Our approach is based on reducing these problems to the emptiness problem of Alternating Büchi Pushdown Systems.

References

1. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T.W., Yannakakis, M.: Analysis of recursive state machines. ACM Trans. Program. Lang. Syst. **27**(4), 786–818 (2005)
2. Alur, R., Chaudhuri, S., Madhusudan, P.: A fixpoint calculus for local and global program flows. In: POPL 2006 (2006)
3. Alur, R., Chaudhuri, S., Madhusudan, P.: Languages of nested trees. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 329–342. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_31
4. Alur, R., Chaudhuri, S., Madhusudan, P.: Software model checking using languages of nested trees. ACM Trans. Program. Lang. Syst. **33**(5), 15 (2011)
5. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_35
6. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63141-0_10
7. Bozzelli, L.: Complexity results on branching-time pushdown model checking. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 65–79. Springer, Heidelberg (2005). https://doi.org/10.1007/11609773_5
8. Burkart, O., Steffen, B.: Model checking the full modal mu-calculus for infinite sequential processes. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) ICALP 1997. LNCS, vol. 1256, pp. 419–429. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63165-8_198
9. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_20
10. Esparza, J., Kucera, A., Schwoon, S.: Model checking LTL with regular valuations for pushdown systems. Inf. Comput. **186**(2), 355–376 (2003)
11. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. Electr. Notes Theor. Comput. Sci. **9**, 27–37 (1997)

12. Kupferman, O., Piterman, N., Vardi, M.Y.: An automata-theoretic approach to infinite-state systems. In: Manna, Z., Peled, D.A. (eds.) Time for Verification. LNCS, vol. 6200, pp. 202–259. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13754-9_11
13. Nguyen, H.-V., Touili, T.: CARET model checking for pushdown systems. In: SAC 2017 (2017)
14. Schwoon, S.: Model-Checking Pushdown Systems. Dissertation, Technische Universität München, München (2002)
15. Song, F., Touili, T.: Efficient CTL model-checking for pushdown systems. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 434–449. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23217-6_29
16. Song, F., Touili, T.: Efficient malware detection using model-checking. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 418–433. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_34
17. Song, F., Touili, T.: LTL model-checking for malware detection. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 416–431. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_29
18. Walukiewicz, I.: Pushdown processes: games and model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 62–74. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61474-5_58

Repair and Generation of Formal Models Using Synthesis

Joshua Schmidt(✉), Sebastian Krings, and Michael Leuschel

Institut für Informatik, Universität Düsseldorf,
Universitätsstr. 1, 40225 Düsseldorf, Germany
joshua.schmidt@uni-duesseldorf.de,
{krings,leuschel}@cs.uni-duesseldorf.de

Abstract. Writing a formal model is a complicated and time-consuming task. Usually, one successively refines a model with the help of proof, animation and model checking. In case an error such as an invariant violation is found, the model has to be adapted. However, finding the appropriate set of changes is often non-trivial.

We propose to partially automate the process by combining synthesis with explicit model checking and implement it in the context of the B method: Guided by examples of positive and negative behavior, we strengthen preconditions of operations or relax invariants of the model appropriately. Moreover, by collecting initial examples from the user, we synthesize new operations from scratch or adapt existing ones. All this is done using user feedback, yielding an interactive assistant. In this paper, we present the foundations of this technique, its implementation using constraint solving for B, and illustrate the technique by synthesizing the formal model of a process scheduler.

1 Introduction

Writing and adapting formal models is a non-trivial task, difficult for beginners and time-consuming even for trained developers. Often, one iterates between changing a model and proof or model checking. Once an error has been detected, the model has to be adapted.

The premise of this paper is that, to some extent, this correction phase can be automated, using negative and positive examples provided by a model checker or by the user. For example, we can synthesize corrected preconditions or invariants in order to repair invariant violations. If deciding to allow an invariant violating state, we know that we need to synthesize relaxed invariants using the given I/O examples. Otherwise, the precondition of the affected operation needs to be strengthened to exclude the state from the model. Moreover, deadlocks can be repaired either by generating a new operation or strengthening the precondition of an existing operation.

When model checking has been exhaustive without finding any invariant violation or deadlock state, we are able to extend the model by synthesizing new

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 346–366, 2018.
https://doi.org/10.1007/978-3-319-98938-9_20

transitions based on state pairs for input and output. In case the machine already contains an operation providing the desired behavior, we relax its precondition or the invariants if necessary. Otherwise, a completely new operation is synthesized.

The tool mainly aims at providing better access to formal methods for beginners. By allowing to define behavior by means of I/O examples the user might be able to learn from the synthesized code. Moreover, if finding an invariant violation or a deadlock state, an automated repair eases the workflow for any user.

In this paper, we present this technique in the context of the B formal method based on our previous publication [32]. In particular, the extensions include:

- extended interactive workflow for the repair of deadlocks and the adaption of existing operations or machine invariants (Sect. 3)
- thorough presentation of the technique, along with support for if-statements and operation parameters (Sect. 4)
- performance improvements due to dynamic expansion of the search space, parallelization, randomized search and symmetry reduction (Sect. 5)
- graphical user interface (Sect. 6)
- performance evaluation (Sect. 8).

2 A Primer on the B-Method

The formal specification language B [1] follows the correct-by-construction approach and is based on first-order-logic and set theory. A formal model in B consists of a collection of machines starting from an abstract specification and successively refining the behavior. The development in B is thus incremental, which increases the maintainability and eases the specification of complex models. In this paper, we always refer to B formal models. The synthesis workflow is applied to a single B machine. A machine consists of variable and type definitions as well as possible initial states. A state is defined by the current evaluation of the machine variables. By defining machine operations, one is able to specify transitions between states. A machine operation has a unique name and consists of B substitutions (aka statements) defining the machine state after its execution, i.e., the values of a set of machine variables are assigned. An operation can have a precondition, allowing or prohibiting execution based on the current state. For instance, a valid machine operation `o` is defined by `o = PRE x>0 THEN x:=x+1 END` using the single assignment substitution of B. Several variables can be assigned either in parallel or in sequence. A state s is called a deadlock if it has no successors, i.e., no operation is enabled. To ensure certain behavior, the user can define machine invariants, i.e., safety properties that have to hold in every reachable state. Hence, the correctness of a formal model refers to the specified properties. In addition to the types explicitly provided by the B language like `INTEGER` or `BOOL`, one can provide user-defined sets. These sets can be defined by a finite enumeration of distinct elements (the set is then referred to as an enumerated set) or left open (called deferred sets). For instance, by defining a set $S = \{s1\}$ the element $s1$ is of type S and can be accessed by name within

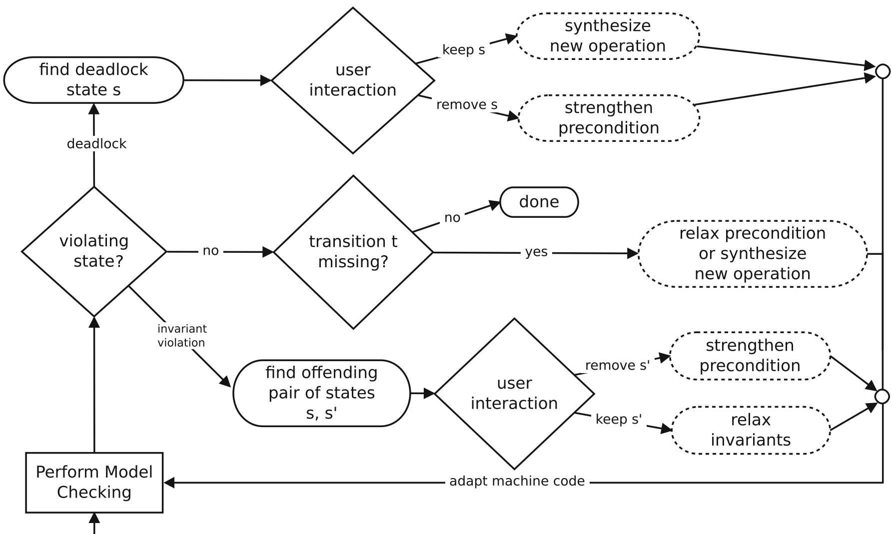


Fig. 1. Interactive workflow to repair and generate formal models using synthesis

the machine. Deferred sets are assumed to be non-empty during proof and also finite for animation.

Using Atelier B [12] or ProB [24–26] one can verify a B model and analyze its state space. In particular, ProB allows the user to animate formal models, providing a model checker and constraint solver. ProB's kernel [24] is implemented in SICStus Prolog [7] using the CLP(FD) finite domain library [8]. Alternatively, a constraint solver based on Kodkod [33] is available [31]. Furthermore, an integration with the SMT solver Z3 [28] can be used to solve constraints [22].

Below, we will focus on classical B [1] for software development, but our approach also works for Event-B and could be extended to other languages supported by ProB such as TLA^+ [23].

3 Interactive Workflow

The process as outlined in Fig. 1 is guided and enforced by ProB. The workflow itself is quite mature and has been fully implemented within the system. Repair is performed successively, that means, we loop until no error can be found anymore and the user is satisfied with the model. Each step starts with explicit model checking performed by ProB. To that effect, the user at least needs to provide a B machine defining variables and an initial state. The dotted nodes mark the parts of the workflow where synthesis is applied. There are three possible outcomes.

First, an invariant violation might be found. We then identify the machine variables that violate the invariants and reduce the examples obtained by the

model checker if possible. The user then can decide to disable the last transition, leading from a state satisfying the invariants to one violating it, by synthesizing a stronger precondition. Alternatively, the system can generate weaker invariants, allowing the violating state.

As a second outcome, the model checker may have uncovered a deadlock state s. The user can then decide between two options:

– Remove s by strengthening the precondition of the involved operation
– Keep s by synthesizing a new operation or adapting an existing one enabling to transition from s to another state *s'*.

Third, the model has been checked and no error was found, that means, model checking was exhaustive or a timeout occurred. We then query the user if state transitions are missing. In case any operation is able to reach the missing states but its precondition is too restrictive, we synthesize a relaxed precondition covering the new state transitions. Otherwise, we synthesize a completely new operation. In general, we need to verify generated programs using the model checker by restarting the workflow. There is no fixed order that determines if an invariant violation or a deadlock state is found first. This depends on the state space and the order of its traversal.

Besides generating an operation from scratch, the user is able to modify an existing operation. The tool initially provides some sample transitions covering the behavior of an operation. This results in providing positive transitions describing the behavior of the operation's substitution. In case the operation provides a precondition, negative transitions are presented describing the behavior of the precondition. The user is then able to provide new transitions either strengthening or relaxing the precondition. Additionally, positive transitions can be provided to modify the substitution of the operation. The machine invariants can be modified in the same manner.

4 Synthesis Technique

The task of (semi-)automatically generating executable programs from a given specification is called program synthesis. There are different approaches in specifying the behavior of a program, for instance, in the form of pre- and postconditions or partial implementations. Jha et al. [19] presented a synthesis technique that uses explicit I/O examples of positive and negative behavior to synthesize loop-free programs that are correct for a set of examples.

In order to restrict the search space, the approach resorts to a library of program components D, each defined in a single static assignment represented by a formula $output = f(inputs)$. For example, for an addition instruction, a constraint would ensure that $o_1 = i_1 + i_2$ holds. Each component is unique and located in a single line of the program. Each line is characterized by its own output variable. The program inputs are also represented as program lines, characterized by their own variable and located in the first lines of the program. The program outputs are located in the last lines of the program overlapping with

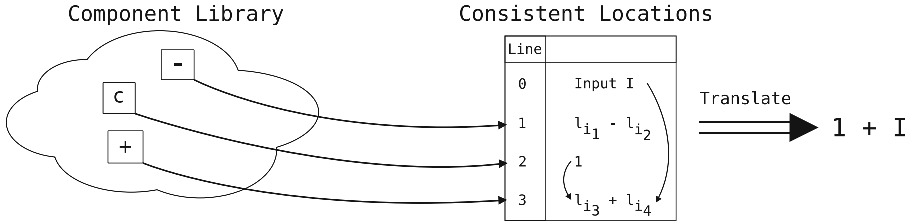


Fig. 2. An example for a possible location mapping between components

components, i.e., the program outputs are defined by compositions of components. In a synthesized program each component is assigned to a unique program line. Given N program inputs, the generated program thus has $M = |D| + N$ lines of code. Each component is used in the generated program but does not necessarily has to participate in the output, i.e., we might generate dead code which is ignored when translating the synthesized program.

Input and output variables are connected using location variables referring to other components. An output location l_{o_k}, $k \in \mathbb{N}$, describes the program line a component is defined in, while an input location l_{i_k} can be interpreted as the line of the program defining its value. Given a set of I/O examples E, synthesis searches for a mapping of locations L between inputs and outputs of components. Afterwards, the locations participating in the output of the synthesized program can be collected and translated to a corresponding abstract syntax tree.

For instance, we search for a program with one input and one output, considering three components describing *addition*, *subtraction*, and an *integer constant*. The set of I/O examples E consists of several examples describing an incrementation of an integer by one. A possible solution is illustrated in Fig. 2, where the solver enumerated the constant c to the value of 1. The single line arrows represent the mapping of the location variables (a solution for L) that participate in the output of the program. Since line one of the program does not participate in the output, the subtraction component is dead code which is ignored when translating the program. In case a program has several outputs, we collect the partial programs from the last lines of the synthesized solution representing the program outputs. Afterwards, the partial programs are combined using the parallel execution substitution of B.

We adapted this technique to synthesize B expressions and predicates using ProB [25,26] as a constraint solver [21]. In order to synthesize B expressions, we use explicit state transitions transforming input to output values and preserving their types. In case of predicates, we replace output states by the evaluation of the desired predicate using the corresponding input states. In this context, an I/O example thus assigns an input state to output either *true* or *false*.

Initially, we are given a set of I/O examples by the user describing the complete desired behavior as well as a list of library components D to be considered during synthesis, which is either prepared automatically or provided by the user. In B, the I/O examples are states of the current machine, i.e., values of machine

variables. The authors refer to P and R as the set of input and output variables of the used library components. Let $I_i \subseteq P$ be the set of input variables of a specific component with the output $O_i \in R$, $1 \le i \le N$. We assume that we derive the formula of the i-th library component using $\Phi_i(I_i, O_i)$. The library is then encoded by the following constraint:

$$\Phi_{lib}(P, R) := \bigwedge_{i=1}^{N} \Phi_i(I_i, O_i)$$

For instance, having two components *addition* and *subtraction* the library is encoded by $(o_1 = i_1 + i_2) \wedge (o_2 = i_3 - i_4)$.

Let E_I be the set of input values and E_O the set of output values of a specific I/O example. Let L be the set of integer valued location variables of inputs and outputs of components $d \in D$. Moreover, L contains locations referring to program parameters, that means, inputs of the overall program. A constraint $\Psi_{wfp}(L, P, R, E_I, E_O)$ defines the control flow of the program to be synthesized and ensures well-formedness. This constraint consists of several parts.

For consistency, output locations are made unique by asserting inequalities between each two locations in R, which is encoded by a constraint $\Psi_{cons}(L, R)$. Component input parameters have to be defined before they are used to prevent cyclic references, which is encoded by

$$\Psi_{acyc}(L, P, R) := \bigwedge_{i=1}^{N} \bigwedge_{I_i \subseteq P, x \in I_i, y \in R, y \equiv O_i} (l_x < l_y). \tag{1}$$

Otherwise, a cyclic expression like $1 + (1 + (\dots))$ would be part of the search space where the location of the right input parameter maps to the addition component itself.

The approach defines program input parameters to be located in the first lines and component outputs in the ensuing lines of a program. To that effect, all components are able to access the program input parameters with respect to the acyclic constraint defined in Eq. (1). Program output parameters are defined in the last lines of the program in order to be able to access all components if necessary. To reduce the overhead, we additionally set each program output parameter to a fixed position by enumerating their positions to one of the last lines, which is achieved by a constraint $\Phi_O(E_O)$. The complete well-definedness constraint is then encoded by

$$\Psi_{wfp}(L, P, R, E_I, E_O) := \bigwedge_{x \in E_I} (0 \le l_x < |E_I|) \wedge \bigwedge_{x \in P} (0 \le l_x < M)$$
$$\wedge \bigwedge_{x \in R} (|E_I| \le l_x < M) \wedge \Psi_{cons}(L, R) \wedge \Psi_{acyc}(L, P, R) \wedge \Phi_O(E_O).$$

We furthermore extend the well-definedness constraint Ψ_{wfp} to ensure well-defined programs according to B. For example, sequences have to be indexed from 1 to n where n is the cardinality of the sequence.

Component inputs can either refer to a program input parameter or another component's output. By setting up constraints for each location, the authors define valid connections between program parameters and components as well as in between components. This includes ensuring type compatibility, that means, only defining connections between locations referring to the same type. We explicitly add constraints preventing connections between differently typed locations to support the ProB constraint solver in finding a solution for the mapping of location variables L. Let $\overline{L} = L_1, .., L_n$, $n > 0$, be a partition of the set of location variables divided by the types they refer to. We then assert:

$$\Psi_{conn}(L) := \forall_{L_1 \in \overline{L}} (\bigwedge_{l_x, l_y \in L_1} l_x = l_y \Rightarrow x = y) \ \wedge \forall_{L_1, L_2 \in \overline{L} \wedge L_1 \neq L_2} (\bigwedge_{l_x \in L_1, l_y \in L_2} l_x \neq l_y)$$

By combining these constraints, the behavior for a single example with a set of inputs E_I and outputs E_O is encoded by

$$\Phi_{func}(L, E_I, E_O) := \exists P, R : \Psi_{wfp}(L, P, R, E_I, E_O) \wedge \Phi_{lib}(P, R) \wedge \Psi_{conn}(L).$$

The overall behavior for a set of examples E containing tuples of input and output is then defined by asserting $\Phi_{func}(L, E_I, E_O)$ for each single example, which is referred to as the behavioral constraint:

$$Behave(E)_L := \bigwedge_{(E_I, E_O) \in E} \Phi_{func}(L, E_I, E_O) \tag{2}$$

When solving the behavioral constraint, we derive an explicit solution for the integer valued location variables in L describing a candidate program satisfying the provided behavior. Afterwards, another semantically different solution $\hat{L}$ is searched by excluding the solution for the location variables L from the behavioral constraint defined in Eq. (2). Of course, we could also use the first solution as is without a further search. However, the user may forget edge cases when providing the set of I/O examples resulting in an ambiguous behavior. We thus want to guide the user to the desired solution as much as possible.

When finding another solution $\hat{L}$, the user chooses among the solutions based on a distinguishing example. That is, a program input where the output of both programs differs, which can be described by

$$\begin{aligned} \exists E_I, E_O, \bar{E}_O : Behave(E)_L \wedge Behave(E)_{\hat{L}} \wedge \Phi_{func}(L, E_I, E_O) \\ \wedge \Phi_{func}(\hat{L}, E_I, \bar{E}_O) \wedge E_O \neq \bar{E}_O. \end{aligned} \tag{3}$$

If no distinguishing example can be found, we assume both programs to be equivalent and choose the smaller one. The system iterates through further solutions in the same fashion. Continuous search for distinguishing inputs provides additional I/O examples, eventually leading to a semantically unique solution. Once found, the synthesized program is returned. It should be noted that searching for another solution possibly results in a solver timeout. In practice, the uniqueness of a synthesized program is therefore only as far as we can decide using the currently selected solver timeout.

During the synthesis of an operation, the user is able to change the output state of a distinguishing example. That means, we do not include an explicit discovered state transition in the set of examples and maybe find it again afterwards. To guarantee unique distinguishing transitions we memorize all results to exclude them from the distinguishing constraint defined in Eq. (3).

When synthesizing explicit if-statement, i.e. using an expression, we have to mix the generation of expressions and predicates leading to a larger search space and a worse performance. To that effect, we also provide an implicit representation of if-statements and implement them as follows: We successively synthesize new operations for each example of a given set of state transitions if necessary, yielding the desired behavior split into several operations. Each operation presents a conditioned block of the statement which is semantically equivalent to explicitly providing if-statements. We start with the first example $(E_I, E_O) \in E$ and synthesize an operation probably with an appropriate precondition. That is, we solve the functional constraint $\Phi_{func}(L, E_I, E_O)$. Afterwards, we decide according to the next example and the so far synthesized operations:

- A previously synthesized operation's substitution fits the current example but the precondition is too restrictive. Hence, we relax the precondition.
- The example can be executed by a synthesized operation. We skip this example since there is nothing to do.
- No operation executes the transition. We generate a new operation only using this example for initialization.

In B, custom types are often accessed via operation parameters. In order to support parameters when synthesizing an operation we add additional components for each custom type. These components are implemented as constants which values are set locally for each example, that means, for each functional constraint of the behavioral constraint defined in Eq. (2). When synthesizing a precondition for an operation that uses parameters, we extend the I/O examples by adding each operation parameter. That means, we view each parameter as a machine variable. The behavioral constraint then considers all necessary information when generating an appropriate precondition. For instance, assuming a machine violates an invariant caused by an operation that uses one parameter and we want to synthesize a strengthened precondition. Furthermore, the machine defines one machine variable. We then extend the states obtained by the model checker by computing the operation parameter for each example and use these extended examples for synthesis.

The synthesis technique by Jha et al. [19] relies on two oracles. The *I/O oracle* is used to define the desired output of the program to be synthesized based on a given input. We replace it by the user. The *validation oracle* is used to check if a synthesized program is correct. To provide it, we apply the synthesized changes to the model and use the ProB model checker for verification.

Moreover, the technique is specialized on synthesis of loop-free programs. However, loops are a special case of the B formal method that are not necessary to be used. A finite loop can be unfolded to several operations providing the

same semantics. In B, one can mistakenly define an infinite loop which, however, is detected by ProB and prevented from execution.

5 Performance Considerations

5.1 Concerned Machine Variables

As we have to consider different combinations of program components, the search space grows exponentially with the number of involved variables. When repairing an invariant violation, we automatically reduce the examples from the model checker to those taking part in the violating state. For instance, assuming we have two machine variables m_1, m_2. Only the variable m_1 is involved in the violated invariant, while m_2 is not. On the one hand, the user can decide to allow the violating state by synthesizing a relaxed invariant. We then only consider the variable m_1 when modifying the machine and do not change any code that involves m_2. However, the synthesized changes may indirectly affect the behavior of the variable m_2. On the other hand, the user can decide to remove the violating state from the model by strengthening the precondition of the operation leading to the violating state. We then additionally consider the machine variables the operation refers to. For instance, if the current precondition of the operation refers to m_2, we consider both variables during synthesis. If synthesis fails, we can consider all machine variables as a last resort. As described in Sect. 4 the *validation oracle*, i.e. the ProB model checker, verifies the modified machine. In B, each operation may access all machine variables. In case of generating an operation from scratch or repairing a deadlock, we cannot draw any conclusions regarding the variables in use. To counter this, we allow the user to mark machine variables that are known not to take part for being skipped.

5.2 Component Library Configuration

Given that B is strictly typed, we are able to reduce the component library to a subset of B. The performance when solving the synthesis constraint itself highly depends on the library configuration. For example, if we do not need arithmetic but only logical operators, the unnecessary operators expand the search space exponentially. To that effect, we consider the types of the variables that are involved in the given examples and only use corresponding operators. Additionally, we statically provide several library configurations for each type. By default, we start with a restricted library configuration to search for simple programs at first, i.e., programs using as few components as possible. In case we do not find a solution, we successively expand the library and restart synthesis. For example, for integers we can start with operators like *addition* and *subtraction*, while not considering constants at first. If this configuration is not sufficient we successively increase the amount of used constants and consider additional operators. When all library configurations failed and no solution can be found using the current timeout, synthesis fails. At this stage, the user may choose to increase the timeout or provide more concise I/O examples.

Of course, in order to synthesize a complex program we probably need to expand the library several times. Therefore, we parallelize synthesis for different library configurations to overcome the loss of performance. In detail, we run $C = |CPU|$ instances of ProB at the same time, where $|CPU|$ is the amount of logical cores that are available to the JVM. All instances have loaded the same model and are always in the same state. When running synthesis, we call the backend C times using distinct library configurations. We listen to the instances and decide as follows for each single instance:

– Success: we return the program and cancel synthesis for the other instances
– Failure: we try another library configuration or do not restart this instance in case there is no library configuration left
– Distinguishing example found: synthesis on this instance is suspended, we present the example to the user and restart the suspended instance after the example has been validated.

To prevent enumerating constants to be synthesized without an upper or lower bound we restrict each constant domain according to the initial examples, which is automatically encoded in the behavioral constraint defined in Eq. (2). As a last resort, we widen such domains if no solution can be found.

5.3 Avoiding Redundancy

To reduce the search space, we implement symmetry reduction for adequate operators. This is done in a preliminary step and directly encoded within the synthesis constraint. Let D be the set of library components and $\bar{D} \subseteq D$ the subset of symmetric components. Assuming n is the amount of input variables of a specific component $d \in D$, we refer to $d(i)$, $i = 1, \ldots, n$, as the i-th input of d. $L(d(i))$ is referred to as the location variable an input $d(i)$ is mapped to whilst $L(d)$ refers to the output location variable of the component. We encode symmetry reduction on the level of operands by the following constraint:

$$\forall d \in \bar{D} : \bigwedge_{i=1}^{n-1} L(d(i)) < L(d(i+1))$$

When considering an *addition* having two inputs i_1 and i_2 this results in $L(i_1) < L(i_2)$. That is, we consider only $o_1 = i_1 + i_2$ and avoid $o_1 = i_2 + i_1$.

Furthermore, we implement symmetry reduction on the level of the same operators to prevent changing the location of components without changing the semantics of the program. This is encoded by the following constraint:

$$\forall d, \bar{d} \in D, d \equiv \bar{d} : L(d) < L(\bar{d})$$

For example, using two components $+_1, +_2$, each representing an addition, this results in $L(+_1) < L(+_2)$, i.e., only $x +_1 y +_2 z$ can be part of the synthesized program while $x +_2 y +_1 z$ can not.

Besides that, the search of a semantically different program for a synthesized solution is another performance bottleneck. There can be numerous equivalent programs to discover, before we are able to determine the uniqueness of a solution using the current timeout or find a program yielding a different semantics. By design, a specific component can only have one output within a synthesized program, and, thus, has to be duplicated if necessary. For instance, a synthesized program uses an union encoded by $o_1 = i_1 \cup_1 i_2$. With respect to this component, synthesis may find a solution mapping a program input parameter p_1 and an enumerated constant to the component inputs, like $p_1 \cup_1 \{1, 2, 3\}$. However, we may need another union with a different output, for instance, to union p_1 with a program input parameter p_2. We then need to use another distinct component $\cup_2$. Given that, the ongoing search possibly results in swapping the inputs of $\cup_1$ and $\cup_2$, i.e., generating $p_1 \cup_2 \{1, 2, 3\}$, but providing the same semantics. Let $L_{out}(d(i))$ be the output location variable that is mapped to the i-th input location of the component d. Given a solution for L, we prevent swapping the inputs of the same operators by asserting the following constraint to hold for the new solution $\hat{L}$:

$$\forall d, \bar{d} \in D, d \equiv \bar{d} : \bigwedge_{i=1}^{n} L_{out}(d(i)) \neq \hat{L}(\bar{d}(i))$$

Unfortunately, the preliminary symmetry reduction is not strong enough to exclude symmetric changes when searching for further solutions. Given the example from above, we assert $l_{i_1} < l_{i_2}$ to hold. This does not prevent the components p_1 and $\{1, 2, 3\}$ to swap locations when searching for another semantically different program, resulting in $\{1, 2, 3\} \cup_1 p_1$ with $l_{i_1} < l_{i_2}$ being satisfied. Moreover, we need to prevent symmetric changes between equivalent components. That means, $\{1, 2, 3\} \cup_2 p_1$ should not be part of the solution. Therefore, we additionally implement a stronger symmetry reduction when searching for further solutions. Let $\bar{D}_{sol}(L)$ be the set of symmetric components that have been used in the solution for L. We assert the following constraints to hold:

$$\forall d \in \bar{D} : \bigwedge_{i=1}^{n} (\bigwedge_{j=1}^{n} L_{out}(d(i)) \neq \hat{L}(d(j)))$$

$$\forall d, \bar{d} \in \bar{D}_{sol}(L), d \equiv \bar{d} : \bigwedge_{i=1}^{n} (\bigwedge_{j=1}^{n} L_{out}(d(i)) \neq \hat{L}(\bar{d}(j)))$$

Given a solution for L, the first constraint ensures that no component output that is mapped to an input of a symmetric component is mapped to any other input of this component. The second constraint ensures the same behavior but in between the same symmetric components, which is only necessary if a symmetric component is included in the component library several times like $+_1, +_2$.

Another factor that has proven to speed up search is to increase variance in synthesized programs and distinguishing examples. To do so, we randomize enumeration order when solving constraints. Using a linear order often causes

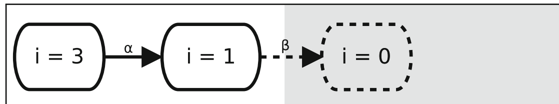


Fig. 3. Abstract visualization of an invariant violation in the user interface

only small syntactical changes among synthesized solutions resulting in more equivalent programs to be generated. Moreover, finding dissimilar distinguishing examples it is less likely to get stuck in some part of the search space where there is no solution [14].

6 User Interface

The graphical user interface is implemented in Java using the JavaFX framework. We use the ProB Java API [4][1] providing an interface to the ProB Prolog kernel to animate and verify formal models and utilize the synthesis backend. The application can be found on Github[2].

When starting the application, the user is able to load a classical B machine. The UI presents a main view split in two areas defining valid and invalid states or transitions. States and transitions are represented as nodes that can be resized and are connected respectively. The workflow starts with explicit model checking as described in Sect. 3. The environment has two different states depending on the result from the model checker: If the model is erroneous, we display the invariant violating trace. Initially, we use a shortened version of the trace containing valid and invalid states. Upon user request, we show further successor and predecessor states. Manually added states are tentative by default and can be validated using ProB. States from the model checker are immutable but can be deleted. The final distribution of the nodes determines the type of the synthesized program as abstracted in Fig. 3. Here, the model checker found an invariant violation in the state `i = 0` which is presented to the user. The states violating an invariant are assumed to be invalid and thus set to be a negative example by default. In the presented setting, the precondition of the operation β will be strengthened to exclude the invariant violating state from the model. If deciding to allow the state by moving it on the side of valid states, the machine invariants will be relaxed. Graphically, a node which state violates an invariant on the side of the pane presenting valid states or vice-versa leads to modifying the invariants. Otherwise, the precondition of the affected operation will be strengthened. If repairing an invariant violation by strengthening a precondition, the considered operation is the one leading to the first state of the trace provided by the model checker that is set to be invalid. During synthesis, distinguishing states may be presented to the user who is asked to place them according to the desired behavior.

[1] The documentation is available online http://www.prob2.de.
[2] https://github.com/joshua27/bsynthesis.

```
MACHINE scheduler
SETS PID = {p1,p2,p3}
VARIABLES  active, ready, waiting
INVARIANT  active <: PID & ready <: PID & waiting <: PID &
           (ready /\ waiting) = {} &            // inv1
           active /\ (ready \/ waiting) = {} &  // inv2
           card(active) <= 1 &                  // inv3
           (active = {}  => ready = {})         // inv4
INITIALISATION active := {} || ready := {} || waiting := {}
OPERATIONS
del(p_PID) = PRE p_PID : PID & p_PID : waiting THEN
                waiting := waiting - {p_PID} END;
new(p_PID) = PRE p_PID : PID & p_PID /: active THEN
                waiting := waiting \/ {p_PID} END
END
```

Fig. 4. A partial scheduler used as a starting point

In case model checking was exhaustive, the user is able to synthesize a new operation. An operation is specified by creating transition nodes and placing them according to the desired behavior. Transition nodes consist of explicit input and output states referring to the variables that should be considered during synthesis. Furthermore, it is possible to modify an existing operation or the machine invariants as described in Sect. 3.

If synthesis succeeds, the generated program is presented to the user who can approve or discard the changes. On approval, the changes are applied to the model followed by a complete run of the ProB model checker. The user interface also presents a list of all B operators that are currently supported by the tool.

7 Example

As an example, we synthesize a B machine managing the states of several processes, which we refer to as a scheduler. Since there are no similar approaches to the semi-automated repair and generation of B formal models we are not able to compare the results. Instead, this example should illustrate the workflow described in Sect. 3.

Initially, we have started from the model shown in Fig. 4 defining only the enumerated set PID containing processes, the three machine variables for the different states of a process and their types as well as the initialization state. We have already synthesized four invariants and two machine operations to create a new process and to delete an existing one from the set of waiting processes. There is no required order and we could have started by generating other operations.

The workflow starts with explicit model checking. Since no errorneous state is found, we proceed to synthesize a new machine operation to activate a waiting task. As a user, we provide six examples which we set to be valid. The machine

```
set_active(p_PID) =
  PRE p_PID : PID & p_PID : waiting & active = {} THEN
        waiting := waiting - {p_PID} || active := {p_PID} END;
set_ready(p_PID) =
  PRE p_PID : PID & p_PID : waiting & active /= {} THEN
       waiting := waiting - {p_PID} || ready := ready \/ {p_PID}
  END;
active_to_waiting = PRE active /= {} & ready = {} THEN
       waiting := waiting \/ active || active := {} END;
ready_to_active(p_PID) =
  PRE p_PID : PID & p_PID : ready & active /= {} THEN
       waiting := waiting \/ active ||
       active := {p_PID} || ready := ready - {p_PID} END
```

Fig. 5. Operations that have been synthesized each at a time

invariant *inv*3 specifies that there can only be one activated process at a time. To that effect, we additionally provide invalid examples for that the operation should block execution, that is, states where either no process is in the set of waiting tasks or there already is an activated task. Synthesis of the substitution using the valid examples succeeds without any further interaction. In contrast to that, the invalid examples do not describe unique behavior. The generation of the precondition thus provides three distinguishing examples that we validate according to the desired behavior. Afterwards, the operation *set_active* shown in Fig. 5 is returned. The operation *set_ready* has been synthesized in the same manner.

When running model checking, a violation of the invariant inv_1 caused by the operation *new* is found, and the user interface presents the trace leading to the violating state. Moreover, the tool automatically decided that only the machine variables *waiting* and *ready* are involved in this invariant violation. We are presented four states that are set to be valid since they do not violate any invariant and one invalid state. We do not change or add any states and run synthesis. This results to synthesizing a strengthened precondition for the operation *new* to remove the invariant violating state from the model. Without any further interaction synthesis terminates and the predicate `p_PID /: ready` is added to the precondition of the operation *new*. Another run of the model checker is exhaustive. We furthermore synthesized two operations to swap a task from being active to either waiting or ready as shown in Fig. 5.

8 Performance Evaluation

In the following we will evaluate the synthesis backend regarding its runtime for several examples. For each program, we provided a complete set of I/O examples describing unique semantics. Consequently, synthesis terminates without any interaction with the user. The synthesized programs $eval_i$ can be found in the

Table 1. An evaluation of the runtime of the synthesis backend

Program	Exact library (in seconds)		Used timeout	Default library (in seconds)		Used timeout	Amount of examples
	no sym.	sym.		no sym.	sym.		
$eval_1$	11.180	2.569	2.5	⊥	18.370	5.0	4
$eval_2$	6.090	0.830	2.5	⊥	57.260	30.0	4
$eval_3$	⊥	9.506	2.5	⊥	⊥	max	5
$eval_4$	⊥	10.670	8.0	⊥	11.320	8.0	6
$eval_5$	⊥	463.860	240.0	⊥	⊥	max	6
inv_1	0.750	0.070	0.5	10.445	9.893	0.5	6
inv_2	⊥	1.630	1.0	433.245	229.340	30.0	8
inv_3	0.054	0.050	0.5	1.775	1.560	0.5	5
inv_4	0.690	0.170	0.5	4.162	2.460	1.0	7
del	0.236	0.230	0.5	1.254	0.929	0.5	6
new	0.943	0.180	0.5	1.925	1.850	0.5	8
new_pre	⊥	0.046	0.5	⊥	2.609	1.0	8
set_active	1.485	0.880	0.5	6.173	4.950	1.0	8
set_ready	3.433	1.010	0.5	9.928	8.540	1.0	9
active_to_waiting	2.964	0.590	0.5	7.135	6.910	1.0	11
ready_to_active	2.792	1.730	1.0	11.459	9.210	1.0	10

Github repository mentioned in Sect. 6. The programs inv_i refer to the invariants of the machine defined in Fig. 4. We will use the average time of ten independent runs using the exact library that needs to be used to synthesize a program and the default library configuration without parallelization as described in Sect. 5.2. We used a maximum solver timeout of 10 min indicated by *max*. $\perp$ indicates a timeout considering the used timeout of a specific benchmark. The used solver timeout and the amount of examples needed to synthesize a certain program are listed in Table 1. Furthermore, we investigate the impact of symmetry reduction suggested in Sect. 5.3. We use the same timeout when synthesizing a program with and without symmetry reduction. All presented times are measured in seconds. The benchmarks were run on a system with an Intel Core I7-7700HQ CPU (2.8 GHz) and 32 GB of RAM.

Amongst other things, the complexity of the synthesis constraint depends on the amount of considered machine variables. However, this also depends on their types which directly affect the components to be considered during synthesis. For instance, if we consider five variables that are all of the same type, it is more complex to find a unique mapping of location variables since the components overlap and can be used at several positions. In case of considering the same

amount of variables but all referring to different types, the possible locations a component can be mapped to are more restricted leading to better performance.

Besides the amount of considered machine variables, the runtime of the synthesis tool also depends on the selected solver timeout. If a solution is found, we search for another semantically different program by excluding the previous solution from the synthesis constraint. On the one hand, this might lead to finding a contradiction. On the other hand, we might need to exhaust the full solver timeout to conclude that we cannot find another solution with the current settings. We then definitely have a runtime higher than the selected timeout. For instance, solving the synthesis constraint for the program $eval_1$ using the exact library that is necessary with symmetry reduction provides a solution after a few milliseconds. Afterwards, we exhaust the solver timeout when searching for another semantically different solution leading to the presented runtime.

When evaluating the impact of symmetry reduction as suggested in Sect. 5.3, one can see that symmetry reduction gains performance for each benchmark. For instance, synthesizing the program inv_1 is around ten times faster using symmetry reduction. Of course, the impact of symmetry reduction also depends on the current settings like the library configuration or the solver timeout.

The program $eval_5$ uses two explicit if-statements so that it is necessary to mix the generation of expressions and predicates. By default, B does not feature if-statements. However, the extended version of B understood by ProB provides an *if-then-else* expression. When synthesizing a program, we use program constructs like expressions backwards. That means, given an output, we search for matching inputs. Of course, ProB is not optimized in doing so for all operators, especially for extensions like if-statements. However, the native B operators are handled efficiently by the ProB constraint solver, which can be seen at the runtimes using the exact library components that are necessary.

When synthesizing the program $eval_2$ or inv_2, we have a large difference between using the exact library and the default library configuration. Of course, this highly depends on the configured library expansions. In this case, the tool at first uses several library configurations that are not sufficient, which is either indicated by finding a contradiction or by exhausting the full solver timeout. Eventually, a sufficient library configuration is found. However, this configuration uses several unnecessary components so that a larger timeout needs to be used leading to the presented runtime. In practice, we parallelize synthesis for different library configurations as described in Sect. 5 leading to better results.

Finally, the performance mostly depends on the amount of program lines which is influenced by the amount of considered library components and program inputs as described in Sect. 4. This can be seen when comparing the runtimes using the exact library with the default library configuration.

9 Related Work

There are many other approaches to program synthesis which could in theory be used to synthesize formal models as well. For instance, techniques to create

divide and conquer algorithms using proof rather than constraint solving [30]. Inductive logic programming [29] is related in the sense that it also starts from positive and negative examples, but is normally not user-guided and is less based on constraint solving but on measures such as information gain. Compared to approaches like [3,9] we synthesize entirely new programs based on input and output values instead of transforming an existing model.

Beyer et al. [6] present a constraint-based algorithm for the synthesis of inductive invariants expressed in the combined theory of linear arithmetic and uninterpreted function symbols. As an input for synthesis, the user specifies a parameterized form of an invariant. In theory, this approach can also be used to synthesize B machine invariants. However, in order to partially automate the development process our workflow is based on explicit model checking providing traces of machine states. Moreover, we do not demand invariants to be inductive and also want to synthesize preconditions and complete operations.

Gvero et al. [15] present a tool to synthesize Java code based on a statistical model derived from existing code repositories. The suggested approach uses natural language processing techniques to accept free-form text queries from the user and infer intentioned behavior from partial or defective Java expressions. The tool learns a probabilistic context-free grammar which is used to generate code. Finally, the user is offered a set of possible solutions ranked by the most frequent uses in the training data. In contrast, we intend to find a unique solution covering exactly the described behavior derived from explicit I/O examples.

In CEGAR [11], spurious counterexamples are used to refine abstractions. Our synthesis tool is guided by real counterexamples and provides an interactive debugging aid for model repair. Moreover, we not only rely on the model checker to find counterexamples but also use ProB as a constraint solver. This leads to more flexibility in model repair and generation, i.e., we can avoid or allow specific states and even extend a machine in case model checking has been exhaustive.

Synthesis can also be applied to functional programming. For instance, Feser et al. [13] present a tool synthesizing functional recursive programs in a λ-calculus. The suggested synthesis approach resorts to a set of higher-order functions as well as language primitives and constants. The tool specializes on synthesizing data structure transformations from explicit I/O examples. The authors define a cost model assigning a positive value to program constructs to find programs with minimal costs using deductive reasoning and a best-first search.

In addition to synthesizing formal models, one can use model checkers and model finders for program synthesis. For instance, Mota et al. [27] use the model finder Alloy [18] to synthesize imperative programs. Programs are described in terms of pre- and post-conditions together with an abstract program sketch defined in Alloy*. To that effect, this approach for specifying programs is more concise than using explicit I/O examples resulting in a smaller search space. While this provides better performance, the user needs more knowledge about the language specification and the program to be synthesized.

10 Future Work

While the example performed in Sect. 7 shows that our approach is feasible in practice, we still have to overcome performance limitations. The B-Method is quite high-level, featuring constructs like sequences, functions or lambda expressions. Powerset construction and arbitrary nesting is allowed as well, affecting the performance of the synthesis tool.

We proposed a default library configuration starting with a small library for each used type and successively considering more components if no solution can be found. In practice, we are not able to efficiently decide for which type the library needs to be expanded. Balog et al. [2] have shown that deep learning techniques can be used to predict the components that are necessary to synthesize a program for a given set of I/O examples, which we also intend to implement for our tool.

One long-term vision would be to combine our approach to model repair with generated models. Clark et al. [10] presented an extension to the internal domain specific language of the ProB Java API which can be used to define classical imperative algorithms. The tool generates an Event-B model describing the algorithm, which can then be processed by ProB and the synthesis tool. If finding an error, we can use our synthesis tool to repair the machine interactively without the need for the end user to know formal methods.

Of course, one could extend our approach to other formal languages such as TLA^+ [23]. As there is an automatic translation of TLA^+ in B [16] and vice-versa [17], we could directly use our implementation inside ProB.

11 Discussion and Conclusion

When enforcing the interactive workflow, model checking is the bottleneck for performance. In order to validate a synthesized program, we need a complete run of the model checker. The performance in discovering a violating state depends on the chosen search strategy as well as the current state of the machine. In order to increase performance in validating synthesized programs it is possible to use distributed model checking [5,20] for models with finite state spaces.

One concern about the suggested approach is that the repeated reparation of a model using generated code affects its comprehensibility and maintainability. Of course, the generated code will be biased to the used library configuration. To counter this, we can use a B simplifier or pretty printer. Moreover, the user could provide short comments for synthesized changes that are added to the code. Besides affecting the code, an automated repair of formal models using synthesis might be considered sceptical since such changes should be made wisely. Using the suggested approach, a synthesized program always fulfils the provided behavior without false positives. As described in Sect. 4, we want to guide the user towards a unique solution as much as possible. However, in practice, we might miss further solutions with a different semantics when searching for a distinguishing example due to a solver timeout. Nevertheless, the user will either

derive a program exactly supporting the described behavior or no solution at all in case synthesis fails. In general, the user should provide an elaborated set of I/O examples each describing different semantics of the desired program, and, in the best case, covering all corner cases that overlap with semantically different programs. Since B is based on states, the representation of system behavior using explicit I/O examples seems to be justified. Nevertheless, the evolution of complex models using synthesis needs to be evaluated in a more detailed way.

Independent from the actually used synthesis technique, we believe that an interactive modelling assistant like the one we outlined above will have its merits both for teaching and for professional use.

References

1. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York, NY, USA (1996)
2. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: DeepCoder: learning to write programs. In: 5th International Conference on Learning Representations (ICLR 2017) (2017)
3. Bartocci, E., Grosu, R., Katsaros, P., Ramakrishnan, C.R., Smolka, S.A.: Model repair for probabilistic systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 326–340. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_30
4. Bendisposto, J., et al.: ProB 2.0 Tutorial. In: Butler, M., Hallerstede, S., Waldén, M. (eds.) Proceedings of the 4th Rodin User and Developer Workshop. TUCS Lecture Notes, vol. 18. TUCS (2013)
5. Bendisposto, J.M.: Directed and distributed model checking of B-Specifications. Ph.D. thesis. Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf (2015)
6. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 378–394. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69738-1_27
7. Carlsson, M., Mildner, P.: Sicstus prolog-the first 25 years. Theory Pract. Log. Program. **12**(1–2), 35–66 (2012)
8. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Glaser, H., Hartel, P., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292, pp. 191–206. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0033845
9. Chatzieleftheriou, G., Bonakdarpour, B., Smolka, S.A., Katsaros, P.: Abstract model repair. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 341–355. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28891-3_32
10. Clark, J., Bendisposto, J., Hallerstede, S., Hansen, D., Leuschel, M.: Generating Event-B specifications from algorithm descriptions. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 183–197. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_11
11. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15

12. ClearSy: Atelier B, User and Reference Manuals. Aix-en-Provence, France (2014). http://www.atelierb.eu/
13. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pp. 229–239. ACM, New York, USA (2015)
14. Gomes, C.P., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. In: Proceedings of AAAI/IAAI, pp. 431–437. American Association for Artificial Intelligence (1998)
15. Gvero, T., Kuncak, V.: Interactive synthesis using free-form queries. In: Proceedings of ICSE, pp. 689–692 (2015)
16. Hansen, D., Leuschel, M.: Translating TLA to B for Validation with ProB. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 24–38. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30729-4_3
17. Hansen, D., Leuschel, M.: Translating TLA$^+$ to B for validation with ProB. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 24–38. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30729-4_3
18. Jackson, D.: Software Abstractions: Logic Language and Analysis. MIT Press, Cambridge (2006)
19. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings ICSE, pp. 215–224 (2010)
20. Körner, P., Bendisposto, J.: Distributed model checking using ProB. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) NFM 2018. LNCS, vol. 10811, pp. 244–260. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77935-5_18
21. Krings, S., Bendisposto, J., Leuschel, M.: From failure to proof: the ProB disprover for B and Event-B. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 199–214. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22969-0_15
22. Krings, S., Leuschel, M.: SMT solvers for validation of B and Event-B models. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 361–375. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_23
23. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA (2002)
24. Leuschel, M., Bendisposto, J., Dobrikov, I., Krings, S., Plagge, D.: From animation to data valiadation: the ProB constraint solver 10 years on Chapter 14. In: Boulanger, J.L. (ed.) Formal Methods Applied to Complex Systems: Implementation of the B Method, pp. 427–446. Wiley ISTE, Hoboken, NJ (2014)
25. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_46
26. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. **10**(2), 185–203 (2008)
27. Mota, A., Iyoda, J., Maranhão, H.: Program synthesis by model finding. Inf. Process. Lett. **116**(11), 701–705 (2016)
28. 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). https://doi.org/10.1007/978-3-540-78800-3_24

29. Muggleton, S., Raedt, L.D., Poole, D., Bratko, I., Flach, P.A., Inoue, K., Srinivasan, A.: ILP turns 20. Mach. Learn. **86**(1), 3–23 (2012)
30. Nedunuri, S., Cook, W.R., Smith, D.R.: Theory and techniques for synthesizing a family of graph algorithms. In: Proceedings First Workshop on Synthesis, EPTCS, vol. 84, pp. 33–46 (2012)
31. Plagge, D., Leuschel, M.: Validating B,Z and TLA$^+$ Using ProB and Kodkod. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 372–386. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_31
32. Schmidt, J., Krings, S., Leuschel, M.: Interactive model repair by synthesis. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 303–307. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_25
33. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_49

Mode-Aware Concolic Testing for PLC Software

Special Session "Formal Methods for the Design and Analysis of Automated Production Systems"

Hendrik Simon(✉) and Stefan Kowalewski

Embedded Software, RWTH Aachen University, Aachen, Germany
{simon,kowalewski}@embedded.rwth-aachen.de

Abstract. During the development of PLC software, standards usually require testing to consider certain coverage criteria. Since a manual generation of coverage tests is tedious and error-prone, automatic approaches as concolic testing are highly desirable. Approaches targeting non-reactive software usually cannot address their peculiarities, e. g. the cyclic execution combined with state-machine behaviour. Hence, we present a novel concolic testing technique to fill this gap. In particular, our technique utilises *operation modes* that typically describe the state machine semantics of single units in PLC programs, also called function blocks. This allows for guiding symbolic execution along paths that conform with the state-machine semantics and are likely to uncover new program behaviour. We show that our technique efficiently generates coverage tests for a variety of programs, outperforming existing approaches tailored to PLC software.

Keywords: Software testing · Integration of formal methods
Programmable Logic Controllers

1 Introduction

Programmable Logic Controllers (PLC) are the predominant control hardware in safety-critical automation applications. As PLCs typically have to interact with their environment with sensors feeding data to inputs and actuators being driven by outputs, logic control software usually resembles a state-machine representing different operation modes. In the area of safety applications, standards as [10] require tests to fulfil certain coverage criteria. Manual generation of test cases, however, quickly becomes costly and error-prone when dealing with larger programs, since engineers usually have to keep track of long sequences of inputs that lead to desired program locations.

Automatic test case generation can significantly lower this burden by deducing coverage tests from the program semantics. One family of techniques to achieve this is Concolic testing, also known as dynamic symbolic execution.

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 367–376, 2018.
https://doi.org/10.1007/978-3-319-98938-9_21

Contribution. In this work we introduce a novel approach to perform concolic testing on PLC software. By introducing awareness for the current operation mode, we present a promising heuristic for guiding the execution through uncovered parts of the program. As a result, we avoid major limitations of related approaches that either limit the application to smaller programs or yield unreliable results due to effectively relying on randomness.

Related Work. Utilising symbolic execution as a tool for test case generation has been an active field of research for decades [6]. Concolic testing, an extension to pure symbolic execution, has been implemented in a variety of tools, e. g. KLEE [5], CREST [4] or LCT [12], most of them focusing on popular programming languages as C or Java. While concolic testing is now productively applied, e. g. to x86 binaries via the tool SAGE [9], the application to logic control software is rare, even in research.

The work of [3] introduces an adaptation of concolic testing to the cyclic execution behaviour of PLCs. Further work [15,16] offers variants of the technique, which are, in contrast to this work, tailored to domain-specific programming languages defined in [11]. The key aspect we will consider in this work is the adherence of PLC software to so called operation modes. This domain-specific feature has already been successfully applied in the context of program verification [2], rendering model checking more efficient. A notion of mode-awareness can also be found in [15], which binds execution paths to mode-representing constructs of the targeted language Sequential Function Chart. However, this notion was not yet used to guide the execution, which will be addressed in this work.

Outline. Sect. 2 will cover the program representation used throughout the paper as well as an introduction to concolic testing. We provide a motivating example in Sect. 3, showing several problems of existing problems and outlining the advantages of mode-awareness. Section 4 presents the main formalism and algorithm, which we evaluated and compared to other approaches in Sect. 5. We finally summarize and conclude our work in Sect. 6.

2 Preliminaries

This section will briefly cover the used program representation, concolic execution and two general concepts for concolic testing of PLC software.

Program Representation. As PLC programs typically comprise several function block instances, which represent stateful functions, we will decompose programs and capture each of these function block instances as a **C**ontrol **F**low **A**utomaton (CFA). A CFA $A = (\mathbf{X}, \mathbf{X}_{in}, V, E, v_e, v_x)$ holds a set of variables $\mathbf{X}$ with $\mathbf{X}_{in}$ being inputs. The set V represents program locations that are connected by edges $e \in E$. Edges further represent instructions, describing the

function blocks behaviour. Finally, frontiers of each function block are captured by an entry- (v_e) and an exit-location (v_x). A program $P = (M, \mathcal{A})$ is defined as a set $\mathcal{A}$ of CFAs with M being the main function's CFA. Note that only M's input variables define the program inputs. As for the instructions, we translate programs into a representation consisting only of *assigns*, *assumes* and *calls.*

Concolic Execution. Concolic execution has the objective to drive the execution of a program along certain paths of interest that are defined by a coverage criterion, e. g. branch coverage. The term concolic means **con**crete and **sym**bolic execution and a wide variety of techniques utilise this concept. The following section aims at giving a general view on the concept.

As we execute programs, we are interested in execution paths, which can be described by a concolic state $c = (\sigma, \rho, \Pi)$. The concrete state is denoted by σ and represents a mapping of variables to concrete values. The symbolic state ρ on the other hand maps variables to logical formulae, describing a set of possible values. The path constraint Π comprises all assumptions made along the path stored as a conjunction of logical formulae, i. e. it stores each branching decision. We implicitly assume that each concolic state is also mapped to a program location, but skip the notation to keep the definitions concise.

Along the execution, a state is modified by instructions. An assignment $x := e$ will have the following impact on the state:

$$\begin{aligned} \sigma &\leftarrow \sigma[x \mapsto val_\sigma(e)] \\ \rho &\leftarrow \rho[x \mapsto val_\rho(e)] \end{aligned}$$

The valuation function $val_{\sigma/\rho}$ evaluates expression e in the context of σ/ρ. Assume statements of the form *assume* e are handled in the following way:

$$\Pi \leftarrow \begin{cases} \Pi \wedge val_\rho(e), & \Pi \wedge val_\rho(e) \text{ satisfiable} \\ false, & \text{else} \end{cases}$$

Note that due to the concrete part of the state, an execution path will always fulfil one of the assumes that resemble a branch in the program. An execution is carried out in this fashion until a desired or maximum depth is reached. It then remains to collect all the branching points on the path and decide, where execution should be driven to next. To this end, an SMT-solver can be used to check whether an alternative path starting at some branching point b is possible or not.

The cyclic behaviour of PLC programs has two major implications for the concolic execution. First, at reaching location v_e of CFA M in cycle i, the program inputs hold new non-deterministic values. Consequently, the symbolic state ρ_j has to be changed to $\rho_{j+1} = \{x \mapsto x_i | x \in M.\mathbf{X}_{in}\}$, assigning each input of M to a fresh symbolic variable. For the concrete state, each program input will be mapped to a random value of the suitable domain. Second, we have to impose a limit on the amount of cycles executed, since PLC programs are designed to run ad infinitum. Alternatively, one could iteratively execute paths one cycle

at a time. Instead of choosing a start point for further execution in terms of a branching point on the current path, one would have to choose from a set of concolic states, deciding which path to execute further. An advantage of this approach is that no cycle limit has to be determined a priori. In the following chapter we will exemplify, what challenges we will face with these approaches.

3 Motivating Example

To pinpoint the problems that concolic testing faces and to motivate our technique, let us take a look at the example in Fig. 1. The graphics captures a snapshot of a program execution in the i-th cycle, part of the program being abstracted and represented by a grey triangle. We assume this part of the program to be complex and contain numerous branching behaviour. The variable m defines the program's state when seen as a state-machine. Let the green edge shown within the triangle, labelled with the condition $m = 1$, be the only uncovered branch left.

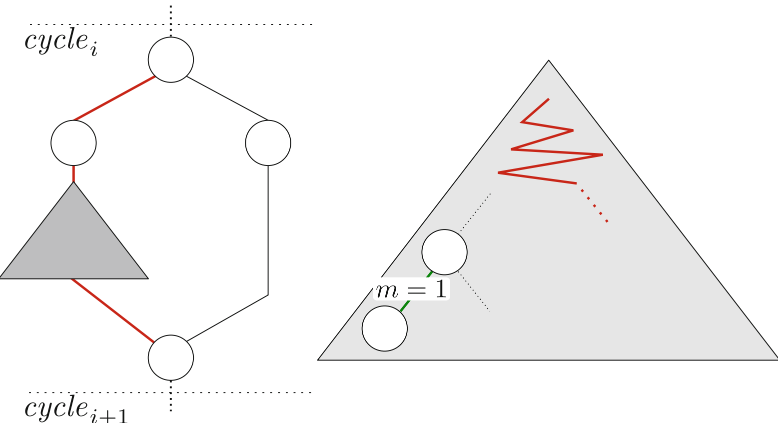


Fig. 1. Snapshot of the execution of a PLC program. The triangle denotes an abstracted part of the program, containing most of the complexity and branching behaviour.

First, assume the cycle-limited approach with the red line being the current execution path. A strategy to find the remaining branch would include choosing one of the many branches within the triangle that is most promising and try to find an execution, leading from the selected to the uncovered branch. Without further guidance, branches in any cycle would represent a good candidate. However, only branches with a concrete state evaluating m to 1, or the chance to set $m := 1$ along the way are suitable candidates to actually cover the missing

(a)

```
(* Variable declarations *)
CASE m OF
0:  IF input = true THEN
        m := 1;
    ELSE
        m := 0;
    END_IF
1:  m := 2;
2:  m := 0;
END_CASE
```

(b)

$m = 0$ → $m = 1$ → $m = 2$ → $m = 0$; $m = 0$ → $m = 0$

Fig. 2. (a) Example program with mode variable m (b) Corresponding modegraph

branch. As a result, selecting the right branch boils down to chance. Even with a sophisticated guidance as, e. g., the CFG-distance based technique from [4] that promotes branches that are close to yet uncovered ones, the problem remains that there are falsely assumed good candidates in each cycle.

Second, assume an iterative approach that has no fixed cycle limit, but executes one cycle at a time, collecting alternative branches as candidates on the way. Again it is hard to decide whether we should either execute an alternative path through a cycle we already stepped through, pick a state that reached a new cycle and execute it for another one, or, if we just have the latter option (we executed all paths up to a current depth), which of these exponentially many states to pick. If we knew that some of these states' concrete state evaluated m to 1, then those were promising candidates.

Investigating those two scenarios, it would be desirable to keep track of the evaluation of m in order to decide where the execution should proceed. This idea will be the foundation of our technique that will keep track of so called *mode-variables* and decide in which direction to execute next based on the evaluation of these variables.

4 Mode Aware Concolic Testing

Our technique is based on the iterative execution variant, i. e. we execute for one cycle at a time. During each cycle, we collect each possible path and store the corresponding states in a priority queue. The key feature to render this feasible is the ordering of the queue, which is done via a distance function utilising mode-variables. Before discussing the algorithm, we will first introduce the necessary formalism for this ordering.

Modespace. Due to the observation that mode-variables implement state-machine behaviour and, thus, typically have a limited set of possible values, we can easily deduce the exact behaviour of these variables. Mode Abstraction

as a technique to compute a modespace has already been proposed by [2]. We will, nevertheless, discuss the most important features of this procedure in the following. Note that we do not provide a method to automatically identify mode-variables, but assume them to be known in advance.

Given a program $p = (M, \mathcal{A})$, we first apply a standard value set analysis (VSA) on M. Note that in order to capture the cyclic behaviour, we add a loop-edge from v_x to v_e in M. The VSA will determine the set of values Y_A that each mode-variable for each function block can be assigned to.

In a second step we want to collect knowledge about the transition of modes, i. e. which modes are reachable from each mode within a cycle. To this end we utilise conditioned slicing [7] on each CFA. The idea of conditioned slicing is to reduce a CFA to the reachable locations given certain constraints on the variables. Consequently, given a CFA A, for each $y \in Y_A$ we perform slicing with the condition $m_A = y$, m_A being A's mode-variable. Performing another VSA on the sliced CFA without loop-edge yields the set $Y_{y\rightarrow}$, containing all values the mode variable can be assigned to in one cycle. As an additional result, slicing this way yields a map $Y_A \mapsto E$ for each CFA A. This map holds the information which edges, and thus, which instruction occur in each mode.

Definition 1 (Modegraph). *A modegraph $G_A = (m_A, Y_A, T, C)$ consists of a mode-variable m_A, a value set Y_A, a transition map $T : Y_A \mapsto 2^{Y_A}$ and an edge containment map $C : Y_A \mapsto 2^E$.*

The procedure above yields a modegraph for each CFA in the program. An example program and its corresponding modegraph are given in Fig. 2a and b, respectively. We call the set $\mathcal{S} = \{G_A | A \in \mathcal{A}\}$ the modespace of program P.

Note that over-approximation during the VSA can lead to imprecise modegraphs, resulting in a less effective guidance.

Mode-Distance. Given a concolic state $c = (\sigma, \rho, \Pi)$ and the modespace $\mathcal{S}$ of program P, we can now approximate how well suited c is for further execution, utilising the knowledge about its mode. To this end, the *modegraph distance* of a concolic state is defined as follows:

$$
\begin{aligned}
sp(y, y', G_A) &= \begin{cases} 0, & y = y' \\ 1 + \min\{sp(y'', y', G_A) | y'' \in T(y)\}, & \text{else} \end{cases} \\
\iota(y, U) &= \begin{cases} \infty, & C(y) \cap U = \emptyset \\ 0, & \text{otherwise} \end{cases} \\
mgd(c, G_A, U) &= \min\{\max\{sp(val_\sigma(m_A), y, G_A), \iota(y, U)\} | y \in Y_A\}
\end{aligned}
$$

```
 1: procedure ModeAwareConcolicTesting(P)
 2:     Q : priorityQueue
 3:     c0 : initial concolic state
 4:     U ← branches(P)                              ▷ initialize U with all branches in P
 5:     S ← computeModeSpace(P)                                   ▷ create the modespace
 6:     Q.insert(c0)
 7:     while ¬(U = ∅) and ¬timeout do
 8:         c = Q.pop()                                       ▷ pick most promising state
 9:         Q.insertAll(executeOneCycle(c), S, U) ▷ insert according to modespace distance
10:         if new branches covered then
11:             U.update()                                ▷ drop covered branches from U
12:             Q.reorder()                       ▷ Distances might have changed, reorder Q
```

Fig. 3. Pseudocode of our mode-aware concolic testing algorithm.

In words, $sp(y, y', G_A)$ calculates shortest path in the modegraph G_A from mode y to mode y'. With U denoting the set of uncovered branches in P, the function $\iota(y, U)$ returns 0 if there are still uncovered branches in mode y, and ∞ otherwise. Hence, the modegraph distance $mgd(c, G_A, U)$ simply takes the current concrete evaluation $val_\sigma(m_A)$ of the mode-variable and computes the shortest path in the modegraph to a mode that still has uncovered branches. Lifting this to modespaces, the *modespace distance* of a concolic state c simply becomes the minimum value of its modegraph distances. Note that this lift could be done differently and in a more sophisticated way. Our goal, however, is to show the potential of this approach even at a simple level.

Testing Procedure. By utilising the modespace distance we now have a mode-aware heuristic that we can use to order the priority queue of concolic states we are dealing with. Our overall testing procedure is shown in Fig. 3. We start by creating an initial concolic state $c_0 = (\sigma_0, \rho_0, true)$, storing random or default values in σ_0 and mapping each program input to a fresh symbolic variable in ρ_0. After creating the modeSpace, the main loop is executed until full coverage is achieved or the time budget is exhausted. In each loop iteration we take the first state in the priority queue $\mathcal{Q}$ and execute it for one cycle. During that cycle on encountering a branch, we try to drive the execution along both paths. While one of those paths is automatically taken due to the evaluation of the concrete state, satisfiability of the other path has to be checked via a call to the SMT solver. Each concolic state generated this way is executed until the end of the cycle and finally sorted into $\mathcal{Q}$. Whenever new branches get covered during this procedure, modespace distances of all states in $\mathcal{Q}$ have to be updated, since $C(y) \cap U = \emptyset$ could now be true for some mode y in one of the modegraphs. Finally, $\mathcal{Q}$ is re-ordered if distances changed. Note that when two states have the same distance, new states are always sorted behind states already in the queue. To avoid exponential usage of memory, we apply an upper limit to the length of $\mathcal{Q}$, keeping only states with highest score.

5 Experiments/Case Study

Our approach was implemented on top of the Arcade.PLC [1] simulation engine, utilising the Z3 SMT Solver [8] for satisfiability checking. Our results were compared to the PLC-tailored concolic testing approach [3] already implemented in Arcade.PLC and the CFG-directed search (CFG-DS) from [4], which we implemented on top of the tool as well. This way, all approaches run in the same framework, allowing for comparison of the techniques without bias caused by differences in e.g. programming language. All experiments were run on an intel-i5 dual-core machine at 2,67 Ghz using 4 logical processors, having 4GB of Ram. The timeout for test case generation was set to 10 min.

We evaluated our approach on a set of function blocks specified by the PLCopen in [13,14], all of them implementing state-machine behaviour with dedicated mode-variables. In addition to this, each of those function blocks makes use of PLC timers and/or edge detection functionality. Each of the time and coverage entries represents the average over 10 runs, since CFG-DS heavily relies on randomly generated paths. All results are listed in Table 1. Note that the time for modespace creation was omitted, since it took approx. 70 ms on average for the small blocks, and 1–2 s for the larger programs, thus, having negligible impact. Further, all calculated modegraphs were precise.

While the merging based technique runs efficiently on the smaller blocks, it already requires from approx. 8 to 13 s for the small but more complex blocks *Modeselector* and *SafelyLimitSpeed*. On the bigger programs, the approach simply times out with an achieved coverage of only 65.5%. This exemplifies a big downside of the merging based technique. Since the states of all paths are always merged at the end of one cycle, the symbolic formulae of the merged state grow exponentially, rendering execution of each further cycle a more difficult task. On the *DiagnosisConcept* program, the algorithm effectively stagnates, spending more and more time on expensive solver calls with each cycle.

The CFG-directed search achieves a better coverage on the larger blocks, while timing out on some of the simpler blocks. Since the CFG-based guidance actually misguides the search in a lot of cases, the algorithm heavily depends on the random number generator choosing a sequence of right input values. This effect especially manifests itself in the test generation for the *Modeselector* block which makes heavy use of Timers and edge detection.

The mode-aware concolic testing on the other hand performs well on each of the smaller blocks, requiring only 2.75 s for the *Modeselector* block. Even for the *DiagnosisConcept*, composed of 4 individual blocks, the algorithm achieves full coverage in under two minutes. Interestingly, on the *SafeMotion* block, albeit achieving high coverage, the technique shows limitations which arise from the simple lifting from modegraph- to modespace distance. Since the behaviour of single blocks in this program is heavily intertwined, the distance function would have to not only minimize over isolated modegraph distances, but rather make these distances dependent on all modes in the program.

Table 1. Case study comparing our approach (MAT) with CFG directed search (CFG-DS) and the merging based technique (MB)implemented in ARCADE.PLC.

Function block	LOC	# branches	Measurements					
			MB		CFG-DS		MAT	
			time[s]	cov. [%]	time[s]	cov. [%]	time[s]	cov. [%]
Antivalent	110	66	1.52	100	1.97	100	**0.89**	100
Equivalent	108	66	1.48	100	7.02	100	**0.46**	100
EmergencyStop	121	70	0.88	100	2.04	100	**0.36**	100
EnableSwitch	136	76	1.78	100	547.66	94.74	**0.51**	100
ESPE	120	70	0.78	100	3.00	100	**0.33**	100
GuardLocking	142	82	1.74	100	8.62	100	**0.48**	100
GuardMonitoring	137	88	2.42	100	90.63	98.86	**0.82**	100
ModeSelector	196	74	7.78	100	TO	82.43	**2.75**	100
SafelyLimitSpeed	142	94	12.94	100	491.82	95.74	**1.16**	100
SafetyRequest	157	92	2.35	100	136.75	98.91	**0.73**	100
Safestop1	162	80	6.91	100	34.50	100	**0.73**	100
TwoHandControlTypeII	132	86	1.40	100	5.55	100	**0.35**	100
TwoHandControlTypeIII	163	112	2.69	100	47.77	100	**0.77**	100
DiagnosisConcept	565	238	TO	65.5	TO	88.4	**116.64**	**100**
SafeMotionIO	961	449	TO	52.27	TO	**87.24**	TO	81.50

6 Conclusion

In this work we have presented a novel approach to concolic testing of PLC software utilising operation mode behaviour. By calculating modespaces, it is possible to obtain a distance measurement for concolic states in order to decide whether or not they yield a good starting point for further program exploration. We were able to show that the proposed technique outperforms related approaches tailored to PLC software as well as other non mode-aware heuristics. As we demonstrated the potential of mode-awareness for this kind of software, it would be desirable for future work to further investigate the lifting from modegraph distance to modespace distance. Since heavily intertwined blocks still pose a problem for the simple approach in this work, a more involved distance measurement for concolic states could be fruitful.

References

1. Biallas, S., Brauer, J., Kowalewski, S.: Arcade.PLC: a verification platform for programmable logic controllers. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE) (2012)
2. Bohlender, D., Kowalewski, S.: Compositional verifcation of PLC software using horn clauses and mode abstraction. In: WODES (2018, to appear)
3. Bohlender, D., Simon, H., Friedrich, N., Kowalewski, S., Hauck-Stattelmann, S.: Concolic test generation for PLC programs using coverage metrics. In: WODES (2016)

4. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: ASE (2008)
5. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI) (2008)
6. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. ACM Commun. **56**(2), 82–90 (2013)
7. Canfora, G., Cimitile, A., De Lucia, A.: Conditioned program slicing. In: Information and Software Technology 40 (1998)
8. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS/ETAPS (2008)
9. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: NDSS (2008)
10. IEC-61508: IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems (2010)
11. John, K.H., Tiegelkamp, M.: IEC 61131–3: Programming Industrial Automation Systems Concepts and Programming Languages. Requirements for Programming Systems, Decision-Making Aids (2010)
12. Kähkönen, K., et al.: LCT: an open source concolic testing tool for Java programs (2011)
13. PLCopen TC5: Safety Software Technical Specification, Version 1.0, Part 1: Concepts and Function Blocks. PLCopen, Germany (2006)
14. PLCopen TC5: Safety Software Technical Specification, Version 1.01, Part 2: User Examples. PLCopen, Germany (2008)
15. Simon, H., Kowalewski, S.: Structural concolic testing for sequential function chart. In: WODES (2018, to appear)
16. Song, J., Jee, E., Bae, D.H.: Automated test sequence generation for function block diagram programs. In: APSEC (2016)

Formalisation of SysML/KAOS Goal Assignments with *B System* Component Decompositions

Steve Jeffrey Tueno Fotso[1,2](✉), Marc Frappier[1], Régine Laleau[2], Amel Mammar[3], and Michael Leuschel[4]

[1] Université de Sherbrooke, GRIL, Sherbrooke, QC, Canada
{Steve.Jeffrey.Tueno.Fotso,Marc.Frappier}@USherbrooke.ca
[2] Université Paris-Est Créteil, LACL, Créteil, France
laleau@u-pec.fr
[3] Télécom SudParis, SAMOVAR-CNRS, Evry, France
amel.mammar@telecom-sudparis.eu
[4] University of Düsseldorf, Düsseldorf, Germany
leuschel@cs.uni-duesseldorf.de

Abstract. The use of formal methods for verification and validation of critical and complex systems is important, but can be extremely tedious without modularisation mechanisms. *SysML/KAOS* is a requirements engineering method. It includes a goal modeling language to model requirements from stakeholder's needs. It also contains a domain modeling language for the representation of system application domain using ontologies. Translation rules have been defined to automatically map SysML/KAOS models into *B System* specifications. Moreover, since the systems we are interested in naturally break down into subsystems (enabling the distribution of work between several agents: hardware, software and human), *SysML/KAOS* goal models allow the capture of assignments of requirements to agents responsible of their achievement. Each agent is associated with a subsystem. The contribution of this paper is an approach to ensure that a requirement assigned to a subsystem is well achieved by the subsystem. A particular emphasis is placed on ensuring that system invariants persist in subsystems specifications.

Keywords: Requirements engineering · Formal models
Modularisation · Goal models · Domain models
SysML/KAOS, *B System*, *Event-B*

1 Introduction

The research work presented in this paper is part of the *FORMOSE* project [5] and focuses on the formal requirements modeling of systems in critical areas such

French National Research Agency (ANR).
Natural Sciences and Engineering Research Council of Canada (NSERC).

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 377–397, 2018.
https://doi.org/10.1007/978-3-319-98938-9_22

as railway or aeronautics. System requirements are modeled using the SysML/KAOS goal modeling language [19]. Translation rules from goal models to *B System* specifications are defined in [24]. They allow the automatic generation of the skeleton of the formal specification of the system [24]. In addition, a language has been defined to express the domain model associated to the goal model, using ontologies [34,36]. Its translation gives the structural part of the *B System* specification [14,37]. Finally, it remains to specify the body of events[1]. Once done, the *B System* specification can be verified, animated and validated using the whole range of tools that support the *B* method [1], largely and positively assessed on industrial projects for more than 25 years [21].

To ensure the distribution of work between several agents and a better maintenability, reusability and scalability of the system, SysML/KAOS allows its partitionning into subsystems: a goal diagram models the main system and further goal diagrams are built for subsystems. Actually, each subsystem is associated with an agent that is responsible for achieving its requirements. The contribution of this paper is an approach to ensure that a requirement assigned to a subsystem is well achieved. The approach uses formal decomposition mechanisms [3] to construct, from the formal specification of a high-level system, the interface of each of its subsystems. The interface of a subsystem describes the requirements that the high-level system expects from the subsystem. Proof obligations are defined to ensure that the invariants of each subsystem is consistent with that of the high-level system. The approach thus ensures that each subsystem achieves its expected goals with respect to constraints set by the high-level system. The proposed approach is illustrated on the *steam-boiler* control specification problem, proposed by *Bauer* in [6].

The remainder of this paper is structured as follows: Sect. 2 briefly describes the SysML/KAOS requirements engineering method and its goal and domain modeling languages, the *B System* formal method, and the translation of SysML/KAOS models. Section 3 presents existing techniques interested in the achievement of system requirements by subsystems, and existing formal decomposition approaches. Finally, Sect. 4 presents our approach illustrated on the steam-boiler control specification problem and Sect. 5 reports our conclusions and discusses future work.

2 Context

2.1 SysML/KAOS Goal Modeling

Presentation. *SysML/KAOS* [19,22] is a requirements engineering method based on *SysML* [16] and KAOS [20]. *SysML* allows for the capturing of requirements and the maintaining of traceability links between those requirements and design deliverables, but it does not define a precise syntax for requirements specification. *KAOS* is a requirements engineering method which allows the representation of requirements to be satisfied by a system and of expectations with

[1] See [13,37] for assessment case studies.

regards to the environment through a hierarchy of goals. Despite of its goal expressiveness, KAOS offers no mechanism to maintain a traceability between requirements and design deliverables, making it difficult to validate them against the needs formulated. In addition, the expression of domain properties and constraints is limited by the expressiveness of UML class diagrams, which is considered insufficient by our industrial partners [5], regarding the complexity and the criticality of the systems of interest. Therefore, for goal modeling, *SysML/KAOS* combines the traceability features provided by *SysML* with goal expressiveness provided by *KAOS*. In addition, *SysML/KAOS* includes a domain modeling language which combines the expressiveness of *OWL* [28] and the constraints of *PLIB* [27].

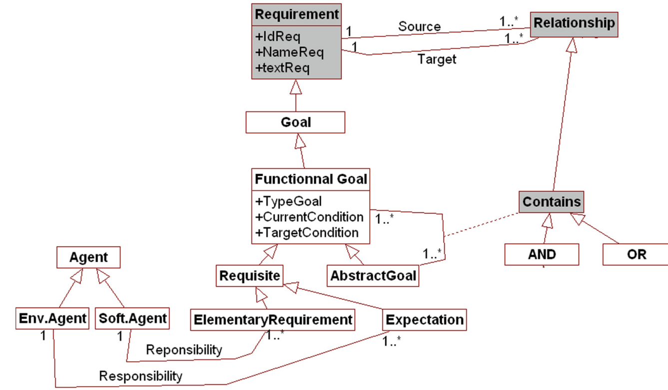


Fig. 1. The SysML/KAOS functional goal metamodel [23]

Presentation: Figure 1 represents the metamodel associated with the modeling of SysML/KAOS functional goals [23]. A goal model consists of goals in relation through operators of which the main ones are: ***AND*** and ***OR***. An ***AND*** *operator* decomposes a goal into subgoals, and all of them must be achieved to realise the parent goal. An ***OR*** *operator* decomposes a goal into subgoals such that the achievement of only one of them is sufficient for the accomplishment of the parent goal. An abstract goal is a functional goal which must be refined. A requisite is a functional goal sufficiently refined to be assigned to an operational agent. Environment agents are responsible of expectations and software agents are responsible of requirements.

Illustration. The challenge of the steam-boiler control specification problem [6] is to specify a system controlling the level of water in a steam-boiler. The system deals with a steam-boiler (SB), a water unit to measure the quantity of water in SB, a pump to provide SB with water, a pump controller and a steam unit to measure the quantity of steam flowing out of SB. In order to be concise, we

limit the system operating modes to the three main ones: *normal*, *degraded* and *rescue*. We also consider two different minimum and maximum water quantities: (*Min1* and *Max1*) for the *normal* mode and (*Min2* and *Max2*), satisfactory levels for the abnormal modes (*degraded* and *rescue*). Figure 3 is a state diagram representing the steam boiler controller operating modes:

- In the *normal* mode, the controller tries to maintain the quantity of water within *Min1* and *Max1*, with all the units behaving correctly. When a failure occurs on the water unit, the mode is set to *rescue*. In case of any other failure, the mode is set to *degraded*.
- In the *degraded* mode, the controller tries to maintain the quantity of water within *Min2* and *Max2*, despite of a possible failure other than a failure of the water unit. If a failure occurs on the water unit, the mode is set to *rescue*. When all failures are repaired, the mode is set to *normal*.
- In the *rescue* mode, the controller tries to maintain the quantity of water within *Min2* and *Max2*, despite of a possible failure of the water unit. It estimates the water quantity, using the measurement of the pump controller and that of the steam unit. When all failures are repaired, the mode is set to *normal*. If the water unit is repaired and there is another failure, the mode is set to *degraded*.

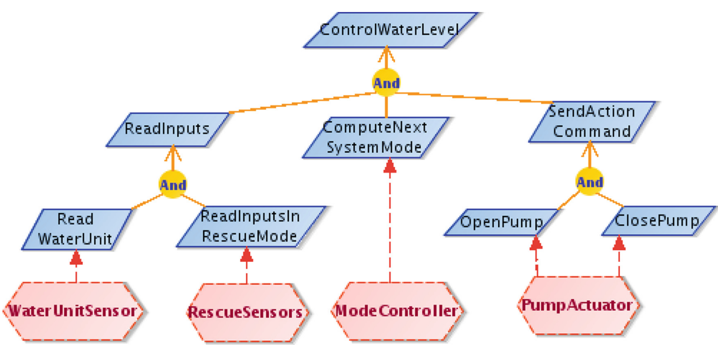


Fig. 2. Excerpt from the steam-boiler control system goal diagram

Figure 2 is an excerpt from the SysML/KAOS goal diagram representing the functional goals of the steam-boiler control system. The main purpose of the system is to control the level of water in the boiler (abstract goal **ControlWaterLevel**). To achieve it, the system must read inputs from the sensors (abstract goal **ReadInputs**), compute the next operating mode using available data (requisite **ComputeNextSystemMode**) and send an action command to the pump (abstract goal **SendActionCommand**). The action may be the opening (requisite **OpenPump**) or the closing (requisite **ClosePump**) of the water pump. To ensure the achievement of goal **ReadInputs**, the system must

be able to obtain *water unit* measurements, in case the *water unit* is behaving correctly (requisite **ReadWaterUnit**). However, since the *water unit* may become defective, the system must also be able to obtain measurements from the *steam unit* and *pump controller*, in order to estimate the quantity of water in the boiler (requisite **ReadInputsInRescueMode**). Four agents are defined for the achievement of requisites: `WaterUnitSensor` responsible of **ReadWaterUnit**, `RescueSensors` responsible of **ReadInputsInRescueMode**, `ModeController` responsible of **ComputeNextSystemMode** and `PumpActuator` responsible of **OpenPump** and **ClosePump**.

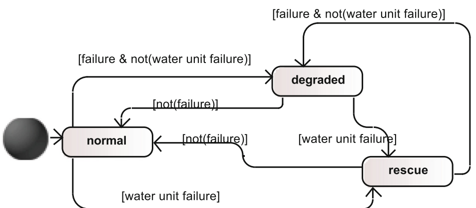


Fig. 3. State diagram of the steam boiler controller operating modes

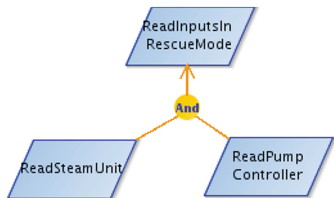


Fig. 4. Excerpt from the goal diagram of the subsystem associated with agent **RescueSensors**

Figure 4 is an excerpt from the SysML/KAOS goal diagram representing the functional goals of the subsystem associated with agent **RescueSensors**. Its main purpose is an abstract goal **ReadInputsInRescueMode** representing the requisite **ReadInputsInRescueMode** of the main system goal diagram (Fig. 2). To achieve it, the system must read values from the steam unit (**ReadSteamUnit**) and pump controller (**ReadPumpController**), in order to estimate the quantity of water in the boiler, in case of a failure of the water unit.

2.2 SysML/KAOS Domain Modeling

Presentation. The SysML/KAOS domain modeling language [34,36] uses ontologies to represent domain models. It is based on *OWL* [28] and *PLIB* [27], two well-known ontology modeling languages. Each domain model corresponds to a refinement level in the SysML/KAOS goal model. The *parent* association represents the hierarchy of domain models. A domain model can define multiple elements. For our purposes, a domain model can define concepts and their individuals, relations, attributes, datasets and predicates [34,36]. A concept represents a collection of individuals with common properties. It can be declared variable ($isVariable = TRUE$) when the set of its individuals can be dynamically updated by adding or deleting individuals. Otherwise, it is constant ($isVariable = FALSE$). A data set represents a collection of data values. A relation captures links between concepts, and an attribute, links between concepts and data sets. They can be variable or constant. Cardinalities are defined

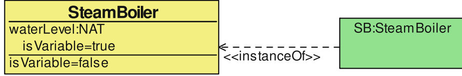


Fig. 5. `steam_boiler_controller_domain_model`: ontology associated with the root level of the goal diagram of Fig. 2

to represent restrictions on relations. A predicate expresses constraints between domain model elements, using the first order logic.

Illustration. Figure 5 represents the SysML/KAOS domain model associated with the root level of the goal diagram of Fig. 2. The steam-boiler entity is modeled as a concept named `SteamBoiler`. As in the case study, adding or deleting a steam-boiler is not considered, property isVariable of `SteamBoiler` is set to `false`. Concept `SteamBoiler` has one individual named `SB`, representing the steam-boiler under the supervision of the system. The attribute `waterLevel` defined in `SteamBoiler` represents the water level in the boiler. It is variable, since it is possible to dynamically change the level of water in the boiler. Refinements of domain model `steam_boiler_controller_domain_model` define the operating mode of the controller using a variable attribute named `operatingMode`, having `SteamBoiler` as domain, and as range, an instance of EnumeratedDataSet containing three data values (`normal`, `degraded` and `rescue`), representing the possible operating modes. For individual `SB`, `operatingMode` is initialised to `normal`, since we consider that the system starts in the normal mode. The associations between a steam-boiler and its sensors and actuators are modeled as relations: a relation named `SteamBoilerSensors` which links the steam-boiler to its sensors and a relation named `SteamBoilerActuators` which links the steam-boiler to its actuators. We have defined three sensors (a steam unit named `SU`, a pump controller named `PC` and a water unit named `WU`) and one actuator (a pump named `P`).

The specification below expresses, using predicates, some domain constraints that have been captured. It should be noted that the variables represent internal states of the controller [26].

```
p2.1: sensorState(WU)= "defective" => operatingMode(SB) ="rescue"
p2.2: (sensorState(WU)="nondefective" & sensorState(SU)=
  "defective")=> operatingMode(SB) ="degraded"
p2.3: (sensorState(WU)="nondefective" & sensorState(PC)=
  "defective")=> operatingMode(SB) ="degraded"
p2.4: (sensorState(WU)="nondefective" & actuatorState(P)=
  "defective")=> operatingMode(SB) ="degraded"
```

Predicate `p2.1` asserts that the operating mode must be *rescue* if the water unit is known to be *defective* and predicates `p2.2`..`p2.4` assert that the operating mode must be *degraded* if a device is known to be *defective*, except for the water unit.

2.3 B System

Event-B is an industrial-strength formal method for *system modeling* [2]. It is used to incrementally construct a system specification, using refinement, and to prove properties. An *Event-B* model includes a static part called context and a dynamic part called machine. Constants, abstract and enumerated sets, and their properties, constitute the static part. The dynamic part includes the representation of the system state using variables constrained through invariants and updated through events. Each event has a ***guard*** and an ***action***. The *guard* is a condition that must be satisfied for the event to be triggered and the *action* describes the update of state variables. A machine can refine another machine, a context can extend other contexts and a machine can see contexts. Proof obligations are defined to prove invariant preservation by events (invariant has to be true at any system state), event feasibility, convergence and machine refinement [2]. *B System* is an *Event-B* syntactic variant proposed by *ClearSy*, an industrial partner in the *FORMOSE* project [5], and supported by *Atelier B* [11]. A *B System* specification consists of components. Each component can be either a system or a refinement and it may define static or dynamic elements. Although it is advisable to always isolate the static and dynamic parts of the *B System* formal model, it is possible to define the two parts within the same component. In the following sections, our *B System* models will be presented using this facility.

2.4 Translation of SysML/KAOS Models

Presentation. The formalisation of SysML/KAOS goal models is detailed in [24]. The proposed rules allow the generation of a formal model whose structure reflects the hierarchy of the SysML/KAOS goal diagram: one component is associated with each level of the goal hierarchy; this component defines one event for each goal. As the semantics of the refinement between goals is different from that of the refinement between *B System* components, new proof obligations for goal refinement are defined in [24]. They complete the classic proof obligations for invariant preservation and for event feasibility. Nevertheless, the generated *B System* specification does not contain the system structure, that are variables, constrained by an invariant, and constants, constrained by properties. This structure is provided by the translation of SysML/KAOS domain models. The corresponding rules are fully described in [35] and their formal verification is described in [14]. In short, domain models identify formal components. A concept without a parent gives a *B System* abstract set. Each concept C, with parent PC, gives a formal constant, subset of the correspondent of PC. Relations and attributes give formal relations. The rules also allow the extraction of the initialisation of state variables.

Illustration. Each refinement level, of the *B System* specification of the steam-boiler control system, is the result of the translation of goal and domain models, except the body of events that are provided manually. The full specification,

```
SYSTEM steam_boiler_controller
  SETS SteamBoiler
  CONSTANTS SB
  PROPERTIES
   axm: SB ∈ SteamBoiler
  VARIABLES waterLevel
  INVARIANT
   inv: waterLevel ∈ SteamBoiler → ℕ
  Event INITIALISATION ≙
   then
     act: waterLevel :∈ {SB} → ℕ
   END
  Event ControlWaterLevel ≙   any wlvl
   where
     grd: wlvl ∈ ℕ
   then
     act: waterLevel(SB) := wlvl
   END
END
```

Fig. 6. Root level of the *B System* specification of the steam-boiler control system

verified using the *Rodin* tool [9], can be found in [37]. Its consistency is ensured with the discharge of *60* proof obligations, *20%* manually and the rest automatically. Interactive proofs were mostly required because of enumerated set definitions that involve partitions: several proof rules require partition rewrites. The generated specification includes three refinement levels.

Figure 6 represents the root level of the *B System* specification of the steam-boiler control system. Concept `SteamBoiler` gives a set and its individual `SB` gives a constant typed with `axm` as an element of set *SteamBoiler*. Attribute `waterLevel` gives a total function from *SteamBoiler* to $\mathbb{N}$ initialised with the action `act` of event `INITIALISATION`. At this level, event `ControlWaterLevel` takes a parameter $wlv \in \mathbb{N}$ and defines it as the level of water in `SB`.

The first refinement level defines a component containing 6 variables, 7 invariants and 4 events (including the *INITIALISATION* event). The second refinement level (`steam_boiler_controller3`) defines a component containing the same set of variables (*waterLevel*, *operatingMode*, *sensorState*, *sensorInput*, *actuatorState*, and *actuatorOutput*), 4 invariants (`p2.1`..`p2.4`) and 6 events: *INITIALISATION*, *ReadWaterUnit*, *ReadInputsInRescueMode*, *ComputeNextSystemMode*, *OpenPump* and *ClosePump*.

The translation rules make it possible to obtain a *B System* specification which becomes complete after the definition of the body of events. The main system is associated with a *B System* model, and each subsystem is associated with another one. However, there are no mechanisms to ensure that subsystem goals are consistent with goals assigned to the high-level system. In the rest of this paper, we are interested in an approach which provides these mechanisms.

3 Existing Work

Section 3.1 presents relevant work related to the assignment of system goals to subsystems, with regard to mechanisms to ensure that subsystem goals are consistent with goals assigned to the high-level system; and Sect. 3.2 presents relevant formal model decomposition approaches.

3.1 Related Work on Goal Assignments

In [10,32,33], approaches are proposed to model a system made of several subsystems. Each subsystem has its own goals (local goals) that are under the

responsibility of an agent (local agent). Each local agent has a degree of freedom in taking local actions to satisfy its local goals. Furthermore, it can negotiate with other agents in attempting to satisfy their local goals. However, to ensure the consistency between subsystem goals and system requirements, a specific subsystem is introduced, under the supervision of a global agent. The global agent focus on the satisfaction of global goals [10]: it can suspend, reschedule or require the execution of an action by a local agent in order to ensure a satisfactorily achievement of system requirements. Although local agents are unaware of objectives of the overall system, they act, under the supervision of the global agent, to ensure the achievement of these overall goals. This approach guarantees a certain degree of freedom in updating the overall goals. However, it requires to implement replanning primitives within local agents and replanning strategies within the global agent.

In [4,15,17], strategies are presented, for a system made of subsystems under the responsibility of agents, to ensure that system requirements are achieved. Each agent evaluates its state and behaviours of other agents, and takes actions that enforce the achievement of system requirements. The decision tree that drives the evaluations made by agents can be internally encoded in each agent, or externally via shared data structures. Algebras are proposed for the representation of desired states. However, relevant changes in the internal structure of an agent require a complete review of established strategies. Furthermore, either the agents must have access to the full behavioral history of other agents, or the strategies must include what agents currently know and what they learn from their actions.

In [12], Wayne only considers subsystems. Each subsystem has its own internal goals and can assign goals to other subsystems. A subsystem can accept or refuse to achieve a goal. Whenever a goal is accepted by a subsystem, the subsystem is responsible to provide feedbacks related to its achievement. The main system can be viewed as a subsystem which does not accept goals while a process can be viewed as a subsystem which does not assign goals. As in [10,33], feedbacks allow one subsystem to monitor the achievement of the goal assigned to another subsystem and to ensure that it is satisfactorily achieved. However, this approach does not take into account the constraints common to goals assigned to different subsystems.

In our approach, formal subcomponents, called interfaces, are extracted from the specification of the high-level system to constrain the specification of subsystem goals. Interface definitions are automatically extracted using a formal model decomposition technique.

3.2 Formal Model Decomposition

Definition. Model decomposition here consists in obtaining, from an initial model, a certain number of less complex models, which can be refined independently and such that the recomposition of subsequent refinement levels produces a model which conforms to the definition of the initial model [3]. We are focussed in the distribution of the elements of the dynamic part of the high-level

system formal specification because the fundamental difference, between two SysML/KAOS agents of the same goal diagram, lies in their behaviors. Recall that a system behaviour is formally represented with a set of events and by all the variables that can be updated by these events and their invariants. The decomposition with respect to the *INITIALISATION* event is trivial and will not be considered. Indeed, whatever the chosen decomposition strategy, a variable ***xx*** assigned to a subcomponent will be initialised within the subcomponent with the same action of the parent component, that initialises ***xx***, since initialisation actions are independent. Similarly, if all events involve only disjoint sets of variables and each invariant involves only the variables appearing in events corresponding to goals of the same agent, the decomposition is trivial: each agent may be assigned a subcomponent defining the events corresponding to the goals assigned to the agent, as well as the associated variables and invariants. The difficulty lies in taking into account variables appearing in events corresponding to goals assigned to different agents (shared variables) and invariants involving variables that are assigned to different subcomponents (shared invariants).

Existing Approaches for the Decomposition of Formal Models. Abrial *et al.* [3] are interested in mechanisms allowing the decomposition of *Event-B* models, and specifically of *Event-B* machines. Indeed, at some point of the refinement process, an *Event-B* machine may have so many events and so many state variables that a further refinement may become difficult or even impossible to manage. Abrial *et al.* consider the decomposition as the distribution of the events of the machine to be split, between several sub-machines. An approach is proposed to handle the variables shared between several events, using external variables and events. Events assigned to a sub-machine are its *internal events*. A variable that is only involved in internal events of a sub-machine is an *internal variable* of the sub-machine. If a variable is involved in internal events of different sub-machines, then it is defined in each of them as an *external variable*. In a sub-machine, an *external variable* can be seen as the input and output channel, allowing the sub-machine to synchronise its activities with other sub-machines defining the same variable. An *external variable* cannot be data-refined. In a sub-machine *A*, an *external event* is an event introduced to simulate the way an external variable is handled, in another sub-machine *B*, by an *internal event* of *B*. External events simulate how external variables are handled in other sub-machines. They do so by abstracting the behaviour of events of the initial machine that involve the external variables. They cannot be refined. Iliasov *et al.* describe in [18] another method for decomposition in *Event-B*. The approach is a special case of the one proposed by Abrial *et al.* restricted to sequential systems for which functionalities can be distributed among several modules.

A decomposition approach, using shared events, is proposed in [7]. It enables the variables of the initial machine to be distributed between sub-machines. When the variables of a global event are distributed between separate sub-machines, each sub-machine defines an event which is a partial version of the global event, and which simulates the action of the global event on the considered

variables. The partial version of an event, defined within a sub-machine, consists in a copy of the original event, restricted to the considered variables (variables of the global event that are allocated to the sub-machine): only parameters, guards and actions referring to the specified variables are preserved from the global event [31].

Silva *et al.* in [31] have identified two methods for decomposition in *Event-B*: the first one considering shared variables and the second one considering shared events. The shared variable decomposition is the decomposition approach introduced in [3] and the shared event decomposition is the one introduced in [7]. A tool is proposed to support the decomposition approaches. For Butler *et al.* [8], the shared event approach is suitable for developing message-passing distributed systems while the shared variable approach is suitable for designing parallel computing programs. Furthermore, it is easier to implement the shared variable approach compared to the shared event approach. Indeed, regarding the shared variable approach, once the events are assigned, the distribution of variables can be done automatically. The decomposition approach is implemented as a plug-in for the *Rodin* platform. The real difficulty lies in the determination of the refinement level from which to introduce the decomposition. Regarding the shared event approach, it may be difficult, once the distribution of variables has been done, to separate the guards and actions of events in order to construct the partial events (a variable cannot appear in two different sub-machines). Regarding invariants, actually, [30,31] let the user select which invariant predicate should be assigned to which subcomponent.

In [29], an approach is proposed for the construction of the specification of an *Event-B* machine from the combination of specifications of several other machines (basic machines). It assumes the partitioning of variables of basic machines, however events can be shared. The machine thus constructed is a composition of basic machines. Proof obligations are proposed in order to verify the composition of machines. The invariant of the composition of machines M_1 to M_n with variables x_1 to x_n respectively is defined as the conjunction of the individual invariants and the composition invariant $I_{CM}(x_1, ..., x_n)$: $I(M_1||...||M_n) \mathrel{\widehat{=}} I_1(x_1) \wedge ... \wedge I_n(x_n) \wedge I_{CM}(x_1, ..., x_n)$. We reuse this definition for the determination of proof obligations associated with the verification of the decomposition of the system specification (Sect. 4.1): the system is seen as a composition of its subsystems.

4 Mechanisms to Ensure the Consistency Between Subsystems and System Requirements

With translation rules, each SysML/KAOS model, whether for the main system or for a subsystem, gives a *B System* specification. To ensure that subsystem goals conform to system requirements, we propose the definition of *B System* components called *interfaces* that will bridge the gap between system and subsystem specifications. An interface of a subsystem defines events that correspond to goals that the system assigns to the subsystem. It also defines variables

involved in these events as well as their constraints. The most abstract level of the formal specification of a subsystem is defined as a refinement of the subsystem interface; this ensures that the subsystem specification conforms to the interface specification. We propose the use of a formal decomposition strategy, applied at the most concrete level of the *B System* specification of the high-level system (parent component), to build subsystem interfaces.

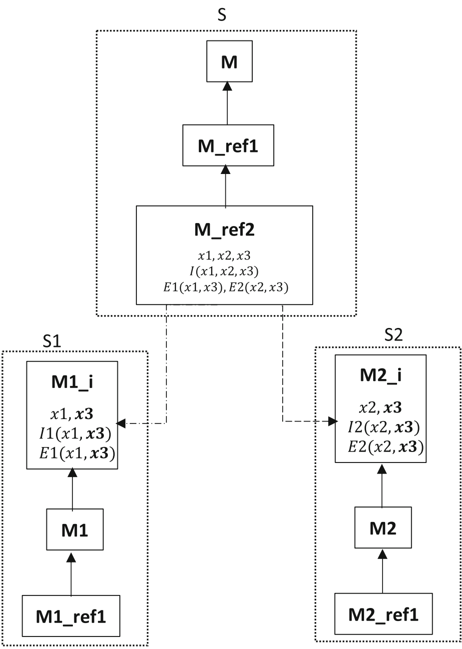


Fig. 7. Illustration of our approach

Figure 7 represents an illustration of our approach for a main system S and two subsystems $S1$ and $S2$. The specification of S defines three components: ***M*** which corresponds to the root level and ***M_ref1*** and ***M_ref2*** which correspond to the first and second refinement levels. The component ***M_ref2*** defines variables *x1, x2* and *x3*, invariant *I(x1, x2, x3)* and events *E1(x1, x3)* and *E2(x2, x3)*. Variable *x3* is shared between the two events. We omitted the corresponding SysML/KAOS goal diagrams; however, the responsibility of *E1* is assigned to $S1$ and the responsibility of *E2* is assigned to $S2$. Thus, the decomposition strategy is used to define interfaces ***M1_i*** for $S1$ and ***M2_i*** for $S2$. The component representing the most abstract level of the specification of each subsystem (***M1*** for $S1$ and ***M2*** for $S2$) is then defined as a refinement of the corresponding interface.

4.1 Construction of Interfaces

Interfaces, Variables and Events. In the SysML/KAOS methodology, goals are assigned to agents. A decomposition of the parent component, based on these assignments, may therefore use the shared variable decomposition approach: each agent gives a formal subcomponent, representing the subsystem interface, and for which the internal events are the correspondences of goals assigned to the agent. For an interface M_i corresponding to agent a_i, internal events of M_i are correspondences of goals assigned to a_i. The variables of M_i are the ones involved in internal events of M_i. If a variable of M_i appears in another interface, then it is an external variable; otherwise, it is an internal variable. Finally, external events are defined in M_i, to emulate how external variables of M_i are handled in other interfaces. Each external event of M_i is an abstraction of an internal event defined in another interface.

Regarding the illustration of Fig. 7, each interface contains the event assigned to the corresponding subsystem (we omitted external events for a sake of clarity). For instance, event *E1(x1, x3)* appears in ***M1_ i***. Variables *x1* and *x3* also appears in ***M1_ i*** because they are involved in *E1(x1, x3)*. Variable *x3* is defined as an external variable in ***M1_ i*** and ***M2_ i***.

Invariants. It remains necessary to decompose the invariants involving variables assigned to different interfaces. Let a component M, containing the variables x_1 and x_2 and the invariant $I(x_1, x_2)$, that is decomposed into subcomponents M_1 containing x_1 without x_2, and M_2 containing x_2 without x_1. Based on the composed invariant defined in [29] (See Sect. 3.2), we advocate that the following conditions are necessary and sufficient, regarding shared invariants (we disregard here properties defined in contexts), in addition to classical requirements of the *Event-B* method [2], to verify the decomposition of M into M_1 and M_2:

- **Subcomponent invariants do not contradict $\mathbf{I(x_1, x_2)}$:** If $I_1(x_1)$ is the invariant introduced in M_1 and $I_2(x_2)$ is the invariant introduced in M_2, then we must prove that: $\exists(x_1, x_2).(I(x_1, x_2) \wedge I_1(x_1) \wedge I_2(x_2))$. Thus, if we consider that x_1 is initialised to $\mathrm{x0}_1$ in M_1 and that x_2 is initialised to $\mathrm{x0}_2$ in M_2 (classical *Event-B* proof obligations ensure that predicate $I_1(x0_1) \wedge I_2(x0_2)$ evaluates to *TRUE*), we must prove that $I(x0_1, x0_2)$ evaluates to *TRUE*. This, in addition, ensures that initialisations in subcomponents preserve the composed invariant.
- Any subsystem event, that can update the value of a variable x introduced in the high-level system, must access x within a mutual exclusion context whenever it is triggered, so that no other event accessing the value of x can be triggered until its termination. Otherwise, it will not be possible to guarantee the accuracy of the value of a variable when an event is triggered within a component. Indeed, within the same *Event-B* component, events are triggered sequentially (to avoid possible inaccuracies in the state of the system), while subcomponents may have parallel behaviors. The constraint thus ensures the

preservation of the sequentiality in the triggering of events coming from the high-level component, with regard to shared variables (the constraint is not necessary for events involving internal variables).

- **Subcomponent events simultaneously preserve global and local invariants:** If an event E_1, introduced in M_1, updates x_1, then we must prove that: $(I(x_1, x_2) \wedge I_1(x_1) \wedge I_2(x_2) \wedge E_1_Guard(x_1) \wedge BA_{E_1}(x_1, x_1')) \Rightarrow (I(x_1', x_2) \wedge I_1(x_1') \wedge I_2(x_2))$. $E_1_Guard(x_1)$ is a predicate denoting that the guard of E_1 is true for the current value of the state variable x_1. BA_{E_1} is the before-after predicate corresponding to E_1[2]. For an event E_2, introduced in M_2, which updates x_2, the proof obligation is: $(I(x_1, x_2) \wedge I_1(x_1) \wedge I_2(x_2) \wedge E_2_Guard(x_2) \wedge BA_{E_2}(x_2, x_2')) \Rightarrow (I(x_1, x_2') \wedge I_1(x_1) \wedge I_2(x_2'))$.

Thus, shared invariants (see Sect. 3.2) can remain in the parent component. It is just necessary to maintain the link between the parent component and the interfaces, through the introduction of a new clause within each interface allowing the referencing of the parent component or through the definition of an external record, and to include the above mentioned proof obligations. The most abstract level of the formal specification of a subsystem is then defined as a refinement of the subsystem interface. It is even possible to add variables, invariants or events in an interface to further constrain the specification of the subsystem or to assign specific goals.

It is also possible to define each variable of an interface as a constant within the others interfaces, where the variable do not appear, and to define shared invariants in each interface. However, this approach carries several difficulties: the update of a shared invariant will have to be done not only within the system specification but also within the specification of each subsystem; and it will be difficult to animate/model-check the formal model, since some variables will be seen as constants. In addition, it will be difficult to ensure that subsystem invariants are always simultaneously preserved, when considering shared variables.

Regarding the illustration of Fig. 7, each interface contains the definition of an invariant. Invariant *I(x1, x2, x3)* remains in ***M_ref2*** and the generated proof obligations are: (1) The invariants defined in ***M1_i*** and ***M2_i*** do not contradict the one defined in ***M_ref2***: $\exists(x1, x2, x3).(I(x1, x2, x3) \wedge I1(x1, x3) \wedge I2(x2, x3))$ (to be satisfied by the initialisation of variables); (2) actions of events *E1(x1, x3)* and *E2(x2, x3)* simultaneously preserve invariants defined in ***M1_i***, ***M2_i*** and the global invariant defined in ***M_ref2***:
(2a) $(I(x1, x2, x3) \wedge I1(x1, x3) \wedge I2(x2, x3) \wedge E1_Guard(x1, x3)$
$\wedge BA_{E1}(x1, x3, x1', x3')) \Rightarrow (I(x1', x2, x3') \wedge I1(x1', x3') \wedge I2(x2, x3'))$;
(2b) $(I(x1, x2, x3) \wedge I1(x1, x3) \wedge I2(x2, x3) \wedge E2_Guard(x2, x3)$
$\wedge BA_{E2}(x2, x3, x2', x3')) \Rightarrow (I(x1, x2', x3') \wedge I1(x1, x3') \wedge I2(x2', x3'))$.

[2] The before-after predicate of E_1 denotes the relationship holding between the state variable of machine M_1 just before (denoted by x_1) and after (denoted by x_1') the triggering of E_1 [2].

Example:

Let M be a component defining invariant $\{x1, x2, x3\} \subset \mathbb{N} \wedge x1 + x2 = x3$ and events $E1 \mathrel{\widehat{=}} then\ x1 := x1 + 1 || x3 := x3 + 1$ and $E2 \mathrel{\widehat{=}} then\ x2 := x2 + 1 || x3 := x3 + 1$. If we consider the decomposition of M into subcomponents *M1* and *M2* with *M1* defining *E1* and invariant $x1 > 100$ and *M2* defining *E2* and invariant $x2 > 100$, the proof obligations are:

(1) $\exists(x1, x2, x3).(\{x1, x2, x3\} \subset \mathbb{N} \wedge x1 + x2 = x3 \wedge x1 > 100 \wedge x2 > 100)$;
(2a) $(\{x1, x2, x3\} \subset \mathbb{N} \wedge x1 + x2 = x3 \wedge x1 > 100 \wedge x2 > 100 \wedge x1' = x1 + 1 \wedge x3' = x3 + 1) \Rightarrow (\{x1', x2, x3'\} \subset \mathbb{N} \wedge x1' + x2 = x3' \wedge x1' > 100 \wedge x2 > 100)$;
(2b) $(\{x1, x2, x3\} \subset \mathbb{N} \wedge x1 + x2 = x3 \wedge x1 > 100 \wedge x2 > 100 \wedge x2' = x2 + 1 \wedge x3' = x3 + 1) \Rightarrow (\{x1, x2', x3'\} \subset \mathbb{N} \wedge x1 + x2' = x3' \wedge x1 > 100 \wedge x2' > 100)$.

They are dischargeable and guarantee that each action of a subsystem preserves not only its invariants, but also invariants of other subsystems and especially the invariant of the high-level system (the subsystems share a variable). They extend the classic proof obligation of invariant preservation [2], which just ensures that each subsystem preserves its own invariants, in the case of several subsystems operating simultaneously to achieve high-level goals and sharing data.

External Events. External events are introduced in interfaces to simulate, in a subsystem, how its external variables are handled in other subsystems. They are proposed by Abrial *et al.* in [3], because no link is maintained between a high-level formal component and its subcomponents after a shared variable decomposition operation. By defining a link between subsystem interfaces and the most concrete component of the high-level system specification, as proposed in Sect. 4.1, it becomes redundant to define external events within interfaces. Through the link between an interface and the parent component, for an external variable x, it would suffice to evaluate the events of the parent component involving x and which are not defined within the interface, to "observe" how x is handled in other subsystems. This approach avoids the difficulties lying in the definition of external events: (1) redundance of the same behavior, associated with an external variable, in each interface where the external variable appears; (2) partitioning of guards and actions of an event to consider only the variables of the interface where the external event must be defined.

4.2 Illustration on the Steam-Boiler Case Study

For the steam-boiler control system, the decomposition must be introduced in the second refinement level (`steam_boiler_controller3`), because it is the most concrete level of the *B System* specification of the main system. Table 1 presents the sharing of state variables between invariants and events of `steam_boiler_-controller3`: variable *waterLevel* is shared between all agents, when considering events where it is involved; variable *sensorState* is shared between agents `WaterUnitSensor`, `RescueSensors` and `ModeController`; and variable *actuatorOutput* is owned by agent `PumpActuator`.

Table 1. Repartition of variables between events and invariants in `steam_boiler_controller3`

Variables	Invariants	Events
waterLevel		*INITIALISATION*, *ReadWaterUnit*, *ReadInputsInRescueMode*, *ComputeNextSystemMode*, *OpenPump*, *ClosePump*
operatingMode	`p2.1`..`p2.4`	*INITIALISATION*, *ReadWaterUnit*, *ComputeNextSystemMode*, *OpenPump*, *ClosePump*
sensorState	`p2.1`..`p2.4`	*INITIALISATION*, *ReadWaterUnit*, *ReadInputsInRescueMode*, *ComputeNextSystemMode*
sensorInput		*INITIALISATION*, *ReadWaterUnit*, *ReadInputsInRescueMode*
actuatorState	`p2.4`	*INITIALISATION*, *ComputeNextSystemMode*, *OpenPump*, *ClosePump*
actuatorOutput		*INITIALISATION*, *OpenPump*, *ClosePump*

Table 2. Overview of interfaces obtained from the decomposition of `steam_boiler_controller3`

Interfaces	Events	Variables
WaterUnitSensor_i	***ReadWaterUnit***, *ReadInputsInRescueMode*, *ComputeNextSystemMode*, *OpenPump*, *ClosePump*	*waterLevel*, *operatingMode*, *sensorState*, *sensorInput*
RescueSensors_i	***ReadInputsInRescueMode***, *ReadWaterUnit*, *ComputeNextSystemMode*, *OpenPump*, *ClosePump*	*waterLevel*, *sensorState*, *sensorInput*
ModeController_i	***ComputeNextSystemMode***, *ReadWaterUnit*, *ReadInputsInRescueMode*, *OpenPump*, *ClosePump*	*waterLevel*, *operatingMode*, *sensorState*, *actuatorState*
PumpActuator_i	***OpenPump***, ***ClosePump***, *ReadWaterUnit*, *ReadInputsInRescueMode*, *ComputeNextSystemMode*	*waterLevel*, *operatingMode*, *actuatorState*, ***actuatorOutput***

We have defined interfaces of the subsystems: each SysML/KAOS agent gives an interface.

Table 2 presents an overview of interfaces obtained from the decomposition of `steam_boiler_controller3`, along with their variables and events. For an interface I, elements in bold are those that are internal to I and the other elements are those that are external (shared). For instance, event ***ReadWaterUnit*** is an

REFINEMENT RescueSensors **REFINES** RescueSensors_i
VARIABLES waterLevel sensorState sensorInput measures
INVARIANT
inv: $measures \in \{SU, PC\} \rightarrow\!\!\!\!\!+ \mathbb{N}$
theorem t1: $ReadSteamUnit_Guard \Rightarrow ReadInputsInRescueMode_Guard$
theorem t2: $ReadPumpController_Guard \Rightarrow ReadInputsInRescueMode_Guard$
theorem t3: $ReadSteamUnit_Post \wedge ReadPumpController_Post \Rightarrow ReadInputsInRescueMode_Post$
Event INITIALISATION $\widehat{=}$ **then**
act1: $waterLevel := \{SB \mapsto Min1\}$ act2: $sensorState := Sensor \times \{nondefective\}$
act3: $sensorInput :\in Sensor \rightarrow \mathbb{N}$ act4: $measures := \emptyset$
END
Event ReadSteamUnit $\widehat{=}$
any wlvl values val1 val2 **where**
grd1: $sensorState(WU) = defective \wedge sensorState[\{SU, PC\}] = \{nondefective\}$
grd2: $\{val1, val2\} \subseteq \mathbb{N}$ grd3: $SU \notin dom(measures)$ grd4: $values = \{SU \mapsto val1\}$
grd5: $wlvl \in \{TRUE \mapsto Min2 .. Max2, FALSE \mapsto \{waterLevel(SB)\}\}(bool(PC \in dom(measures)))$
then
act1: $waterLevel(SB) := wlvl$ act2: $sensorInput := sensorInput \lhd\!\!- values$
act3: $measures := \{TRUE \mapsto \emptyset, FALSE \mapsto values\}(bool(PC \in dom(measures)))$
END

Fig. 8. Overview of the root level of the *B System* specification of the subsystem `RescueSensors`

internal event in interface *WaterUnitSensor_ i*, while event *ReadInputsInRescueMode* is an external event that simulates, in *WaterUnitSensor_ i*, the behaviour of internal event ***ReadInputsInRescueMode***, defined in interface *RescueSensors_ i*; variable ***actuatorOutput*** is an internal variable in interface *PumpActuator_ i*, while variable *waterLevel* is an external variable. Variable *waterLevel* is defined as an external variable in all interfaces because it is involved in internal events of the four interfaces. In addition, since variable *waterLevel* is involved in all events, each interface defines an external event that simulates the behaviour of each event not internal to the interface. Once it will be possible to define a link between each interface and its parent component, we believe that it will no longer be necessary to define these external events. Invariants `p2.1`..`p2.4` remain in `steam_boiler_controller3`; however, if needed, invariants `p2.1`..`p2.3` can be defined in interfaces *WaterUnitSensor_ i*, *ModeController_ i* and *PumpActuator_ i*, and invariant `p2.4` can be defined in *ModeController_ i*.

Figure 8 is an overview of the root level of the *B System* specification of the subsystem associated to agent `RescueSensors`. It is a refinement of interface *RescueSensors_ i*. We provide the specification of the event corresponding to goal **ReadSteamUnit** of the goal diagram of Fig. 4: when water unit `WU` is defective and steam unit `SU` and pump controller `PC` are non-defective (`grd1`), then a natural integer `val1` is set as the input obtained from sensor `SU` (`act2`). Controller variable *measures* is used to take into account the *non-simultaneity* and the *non scheduling* of the measurement of values of sensors `SU` and `PC`, introduced in the goal diagram with the use of the *AND* operator between the root and first refinement levels. Within event **ReadSteamUnit**, variable *measures* allows the controller to consider the following cases: (1) when the measurement of values of `SU` and `PC` has not yet been achieved ($SU \notin dom(measures) \wedge PC \notin dom(measures)$), the value of `SU` is measured (`grd4`) and saved into variables *sensorInput* (`act2`) and *measures* (`act3`); (2) when

the value of PC has already been measured, the value of SU is measured and used, together with the value of PC, to estimate the water level (grd4 and act1). Action act3 allows, regarding the last case, to reset the content of variable *measures* for further measurements. The behavior of event **ReadPumpController** is identical to that of **ReadSteamUnit**, except that it performs the measurement of the value of PC.

Interface *RescueSensors_i* provides variables *waterLevel*, *sensorState* and *sensorInput* and event **ReadInputsInRescueMode** to component *RescueSensors*. Theorems t1..t3 are defined to express SysML/KAOS proof obligations related to the use of the *AND* operator[3] (Fig. 4) [24].

4.3 Discussion

The proposed approach uses the shared variable decomposition strategy and proof obligations to ensure that subsystems specifications conform to system requirements. The approach fits into the following process which is applicable for any system *S* made of subsystems *S1..Sn*, assuming that SysML/KAOS models of *S*, *S1..Sn* are already defined:

(1) Translate SysML/KAOS models of *S* into a *B System* specification made of components $C_S_0, C_S_1, ..., C_S_p$, where C_S_r is a refinement of C_S_{r-1} (Sect. 2.4 and [14,24,35]).
(2) Complete the specification obtained from (1) by specifying the body of events (Sect. 2.4 and [13]).
(3) Use the formal decomposition strategy to construct, from C_S_p, the formal subcomponents $S1_i..Sn_i$, where *Sk_ i* denotes the interface of subsystem *Sk*, containing the specification of goals that *S* assigns to *Sk* with their associated variables and constraints (Sect. 4.1).
(4) For each subsystem *Sk*:
   (i) **IF** *Sk* is made of subsystems
   THEN restart the whole process with *Sk* as the high level system
   ELSE apply steps (1) and (2) on *Sk*
   (ii) Set component C_Sk_0 as a refinement of *Sk_ i*.

The approach makes it possible to independently define, check and evolve the specifications of subsystems. It also allows centralised updates of constraints and goals assigned to subsystems: global update of the high-level system specification, which can be automatically propagated into interfaces, and/or local update of an interface, which is available for the whole subsystem specification.

5 Conclusion and Future Work

This paper focusses on an approach to ensure that a requirement assigned to a subsystem is well achieved by the subsystem. The approach uses a formal model

[3] For an event G, G_Guard represents the guard of G and G_Post represents the post condition of its actions [24].

decomposition strategy and proof obligations to guarantee that subsystem goals are consistent and meet system requirements expressed in SysML/KAOS models that are translated to *B System* specifications. The approach is appraised on the *steam-boiler* control specification problem [6], using *Rodin* [9], an industrial-strength tool supporting the *Event-B* method. Its advantages are discussed, with regard to some relevant related work.

Work in progress is aimed at studying the back propagation of updates on a *B System* specification within the associated SysML/KAOS model. We are also working on integrating the approach within the open-source platform *Openflexo* [25] which federates the various contributions of *FORMOSE* project partners [5].

Acknowledgment. This work is carried out within the framework of the *FORMOSE* project [5] funded by the French National Research Agency (ANR). It is also partly supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
2. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, Cambridge (2010)
3. Abrial, J.R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: application to Event-B. Fundamenta Informaticae **77**(1–2), 1–28 (2007)
4. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM (JACM) **49**(5), 672–713 (2002)
5. ANR-14-CE28-0009: Formose ANR project (2017). http://formose.lacl.fr/
6. Bauer, J.C.: Specification for a software program for a boiler water content monitor and control system. Institute of Risk Research, University of Waterloo (1993)
7. Butler, M.: Synchronisation-based decomposition for Event-B. RODIN Deliverable D19 Intermediate report on methodology, pp. 47–57 (2006)
8. Butler, M.: An approach to the design of distributed systems with B AMN. In: Bowen, J.P., Hinchey, M.G., Till, D. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 221–241. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0027291
9. Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.): Rigorous Development of Complex Fault-Tolerant Systems. LNCS, vol. 4157. Springer, Heidelberg (2006). https://doi.org/10.1007/11916246
10. Claassen, A.F., Lathon, R.D., Rochowiak, D.M., Interrante, L.D.: Active rescheduling for goal maintenance in dynamic manufacturing-systems. In: Proceedings of the AAAI Spring Symposium (1994)
11. ClearSy: Atelier B: B System (2014). http://clearsy.com/
12. Davis, W.J.: Evaluating performance of distributed intelligent control systems. NIST Special Publication SP, pp. 225–232 (2001)
13. Tueno Fotso, S.J., Frappier, M., Laleau, R., Mammar, A.: Modeling the hybrid ERTMS/ETCS level 3 standard using a formal requirements engineering approach. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) ABZ 2018. LNCS, vol. 10817, pp. 262–276. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91271-4_18

14. Tueno Fotso, S.J., Mammar, A., Laleau, R., Frappier, M.: Event-B expression and verification of translation rules between SysML/KAOS domain models and B system specifications. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) ABZ 2018. LNCS, vol. 10817, pp. 55–70. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91271-4_5
15. Goranko, V.: Coalition games and alternating temporal logics. In: Proceedings of the 8th Conference on Theoretical Aspects of Rationality and Knowledge, pp. 259–272. Morgan Kaufmann Publishers Inc. (2001)
16. Hause, M., et al.: The SysML modelling language. In: Fifteenth European Systems Engineering Conference, vol. 9. Citeseer (2006)
17. Van der Hoek, W., Wooldridge, M.: Multi-agent systems. Found. Artif. Intell. **3**, 887–928 (2008)
18. Iliasov, A., et al.: Supporting reuse in Event B development: modularisation approach. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 174–188. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11811-1_14
19. Laleau, R., Semmak, F., Matoussi, A., Petit, D., Hammad, A., Tatibouet, B.: A first attempt to combine SysML requirements diagrams and B. Innov. Syst. Softw. Eng. **6**(1–2), 47–54 (2010)
20. van Lamsweerde, A.: Requirements Engineering - From System Goals to UML Models to Software Specifications. Wiley (2009)
21. Lecomte, T., Deharbe, D., Prun, E., Mottin, E.: Applying a formal method in industry: a 25-year trajectory. In: Cavalheiro, S., Fiadeiro, J. (eds.) SBMF 2017. LNCS, vol. 10623, pp. 70–87. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70848-5_6
22. Mammar, A., Laleau, R.: On the use of domain and system knowledge modeling in goal-based Event-B specifications. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 325–339. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_23
23. Matoussi, A.: Building abstract formal Specifications driven by goals. Ph.D. thesis, University of Paris-Est, France (2011). https://tel.archives-ouvertes.fr/tel-00680736
24. Matoussi, A., Gervais, F., Laleau, R.: A goal-based approach to guide the design of an abstract Event-B specification. In: ICECCS 2011, pp. 139–148. IEEE Computer Society
25. Openflexo: Openflexo project (2018). https://github.com/openflexo-team/
26. Parnas, D.L., Madey, J.: Functional documents for computer systems. Sci. Comput. Program. **25**(1), 41–61 (1995)
27. Pierra, G.: The PLIB ontology-based approach to data integration. In: Jacquart, R. (ed.) Building the Information Society. IIFIP, vol. 156, pp. 13–18. Springer, Boston, MA (2004). https://doi.org/10.1007/978-1-4020-8157-6_2
28. Sengupta, K., Hitzler, P.: Web ontology language (OWL). In: Encyclopedia of Social Network Analysis and Mining, pp. 2374–2378 (2014)
29. Silva, R.: Towards the composition of specifications in Event-B. Electron. Notes Theor. Comput. Sci. **280**, 81–93 (2011)
30. Silva, R., Butler, M.J.: Shared event composition/decomposition in Event-B. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 122–141. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_7
31. Silva, R., Pascal, C., Hoang, T.S., Butler, M.J.: Decomposition tool for Event-B. Softw. Pract. Exper. **41**(2), 199–208 (2011)

32. Suski, G., et al.: The nova control system-goals, architecture, and system design. In: Distributed Computer Control Systems 1982, pp. 45–56. Elsevier (1983)
33. Tan, J.K.: Health management information systems: methods and practical applications. Jones & Bartlett Learning (2001)
34. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: Towards using ontologies for domain modeling within the SysML/KAOS approach. In: RE Workshops, pp. 1–5. IEEE Computer Society (2017)
35. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: Formal Representation of SysML/KAOS Domain Models. ArXiv e-prints, cs.SE, 1712.07406, December 2017. https://arxiv.org/pdf/1712.07406.pdf
36. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: The SysML/KAOS Domain Modeling Approach. ArXiv e-prints, cs.SE, 1710.00903, September 2017. https://arxiv.org/pdf/1710.00903.pdf
37. Tueno, S., Laleau, R., Mammar, A., Frappier, M.: The SysML/KAOS Domain Modeling Language (Tool and Case Studies) (2017). https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master

Analysing AWN-Specifications Using mCRL2 (Extended Abstract)

Rob van Glabbeek[1,2], Peter Höfner[1,2(✉)], and Djurre van der Wal[1,3]

[1] Data61, CSIRO, Sydney, Australia
Peter.Hoefner@data61.csiro.au
[2] Computer Science and Engineering, University of New South Wales, Sydney, Australia
[3] Formal Methods and Tools, University of Twente, Enschede, The Netherlands

Abstract. We develop and implement a translation from the process Algebra for Wireless Networks (AWN) into the milli Common Representation Language (mCRL2). As a consequence of the translation, the sophisticated toolset of mCRL2 is now available for AWN-specifications. We show that the translation respects strong bisimilarity; hence all safety properties can be automatically checked using the toolset. To show usability of our translation we report on a case study.

1 Introduction

The Algebra for Wireless Networks (AWN) [11] is a variant of classical process algebras that has been particularly tailored to model and analyse protocols for Mobile Ad hoc Networks (MANETs) and Wireless Mesh Networks (WMNs). Among others it has been successfully used to model and analyse the Ad hoc On-Demand Distance Vector (AODV) routing protocol [30], one of the most popular protocols widely used in WMNs. [12,16]

AWN provides the right level of abstraction to model key features of protocols for (wireless) networks such as unicast and broadcast for message sending, while abstracting from implementation-related details. It is equipped with a (completely unambiguous) formal semantics, which is given in form of structural operational semantics rules. These rules generate a transition system that can be used to describe the behaviour of a protocol.

The algebra has been integrated in the interactive proof assistant Isabelle/HOL [6]. This enabled, amongst others, the machine-checked verification of key correctness properties of AODV. However, apart from that there is only little tool-support for AWN. To provide automatic analysis for protocols written in AWN, the algebra has been used in combination with the model checker Uppaal. [10] The input model for Uppaal [1,2]—a network of timed automata—was created manually and the correctness of this model needed to be established manually as well.

In sum, AWN falls short when it comes to automated analysis of specifications. The development of special-purpose tools for AWN is cumbersome,

© Springer Nature Switzerland AG 2018
C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 398–418, 2018.
https://doi.org/10.1007/978-3-319-98938-9_23

error-prone and time consuming. Hence we follow the general approach to make use of highly sophisticated off-the-shelf tools that offer high-performance analysis. In this paper we present and implement an automatic translation from AWN into the milli Common Representation Language (mCRL2) [18].

mCRL2 is a formal specification language with an associated collection of tools offering support for model checking, simulation, state-space generation, as well as for the optimisation and analysis of specifications. [8] The toolset has been used in countless case studies, including the analysis software for the CERN's Large Hadron Collider [21], and the IEEE 1394 link layer [25].

We do not only develop an automatic translation from AWN to mCRL2, which allows us to use the mCRL2 toolset for any protocol specification written in AWN, we also show that the transition system induced by an AWN-specification and the transition system in mCRL2 that stems from our translation are strongly bisimilar. As a consequence, any safety property that has been (dis)proven in the mCRL2 setting also holds/does not hold for the original specification, written in AWN. To illustrate the usefulness of our translation, we report on a case study that analyses the Ad hoc On-Demand Distance Vector (AODV) routing protocol [30], which we formalised in AWN before [12,16].

2 The Algebra for Wireless Networks

The Algebra for Wireless Networks (AWN) [10,12] is a variant of standard process algebras (e.g. [3,5,20,26]) particularly tailored for (wireless) protocols: it defines the protocol in a pseudo-code that is easily readable, and provides the right level of abstraction to model key protocol features.

The algebra offers a local broadcast mechanism and a conditional unicast operator—allowing error handling in response to failed communications while abstracting from link layer implementations of the communication handling—and incorporates data structures with assignments. As a consequence it allows to describe the interaction between nodes in a network with a dynamic or static network topology, and hence is ideal to describe all kinds of protocols.

AWN comprises five layers: *sequential processes* for encoding the protocol as a recursive specification; *parallel composition* of sequential processes for running multiple processes simultaneously on a single (network) node; *node expressions* for running (parallel) processes on a node while tracking the node's address and all nodes within transmission range; *partial network expressions* for describing networks as parallel compositions of nodes and allowing changes in the network topology, and *complete network expressions* for closing partial networks to further interactions with the environment.

Due to lack of space we cannot present the full syntax and semantics of AWN, which can be found in [11]. Table 1 summarises the syntax. In AWN a network is modelled as an encapsulated parallel composition of network nodes (Lines 14 and 15 in Table 1). An individual node has the form $i:P:R$, where i is the unique identifier of the node, P characterises the process running on the node, and the set R contains all identifiers of nodes currently in transmission range of

Table 1. Process expressions

$X(exp_1, \ldots, exp_n)$	Process name with arguments
$P + Q$	Choice between processes P and Q
$[\varphi]P$	Conditional process
$[\![\mathtt{v} := exp]\!]P$	Assignment followed by process P
broadcast$(ms).P$	Broadcast ms followed by P
groupcast$(dests, ms).P$	Iterative unicast or multicast to all destinations $dests$
unicast$(dest, ms).P \blacktriangleright Q$	Unicast ms to $dest$; if successful proceed with P; otherwise Q
send$(ms).P$	Synchronously transmit ms to parallel process on same node
deliver$(data).P$	Deliver data to client (application layer)
receive$(\mathtt{m}).P$	Receive a message
ξ, P	Process with valuation
$P \langle\!\langle Q$	Parallel processes on the same node
$i : P : R$	Node i running P with range R
$N \Vert M$	Parallel composition of nodes
$[N]$	Encapsulation

i—the nodes that can receive messages sent by i. On each node several processes may be running in parallel (Line 12 in Table 1). A sequential process is given by a sequential process expression P, together with a *valuation* ξ associating values $\xi(\mathtt{v})$ to variables $\mathtt{v}$ maintained by this process (Line 11).

AWN uses an underlying data structure with several types, variables ranging over these types, operators and predicates. Predicate logic yields terms (or *data expressions*) and formulas to denote data values and statements about them. The choice of this data structure is tailored to any particular application of AWN. It must contain the types `DATA`, `MSG`, `IP` and $\mathcal{P}(\mathtt{IP})$ of application layer data, messages, IP addresses—or other node identifiers—and sets of IP addresses.

In addition, AWN employs a collection of *process names*, each carrying parameters of various types. Every process name X comes with a *defining equation* $X(\mathtt{v}_1,\ldots,\mathtt{v}_n) \stackrel{def}{=} P$, in which each $\mathtt{v}_i$ is a variable of the appropriate type and P a sequential process expression.

Lines 1 to 10 describe sequential process expressions. $X(exp_1, \ldots, exp_n)$ is a call to the process defined by the process name X, with expressions of the appropriate types substituted for the parameters. $P + Q$ may act either as P or as Q, depending on which of the two processes is able to act. If both are able to act, a non-deterministic choice is made. Given a valuation of the data variables by concrete data values, the sequential process $[\varphi]P$ acts as P if φ evaluates to true, and deadlocks otherwise. In case φ contains free variables, values are assigned to these variables in any way that satisfies φ, if possible. The process $[\![\mathtt{v} := exp]\!]P$ acts as P, but under an updated valuation of the data variable $\mathtt{v}$. The process **broadcast**(ms) broadcasts ms to the other network nodes within transmission range, and subsequently acts as P; **unicast**$(dest, ms).P \blacktriangleright Q$ is a process that tries to unicast the message ms to the destination $dest$; if successful it continues to act as P and otherwise as Q. It models an abstraction of an acknowledgment-of-receipt mechanism. The process **groupcast**$(dests, ms).P$ tries to transmit ms

Process 1 Voting

```
Voting(lip,lno,voted,ip,no) ≝
 1. (receive(m) . [m = B(sip, sn)]    /* receive ballot */
 2.    Eval(sip,sn,lip,lno,voted,ip,no))
 3. + [ !voted ]       /* cast a ballot */
 4.    ((broadcast(B(ip, no)) .
 5.        Eval(ip,no,lip,lno,true,ip,no))
 6.      + (receive(m) . [ m = B(sip, sn) ]
 7.        Eval(sip,sn,lip,lno,voted,ip,no)) )
```

Process 2 Vote Evaluation

```
Eval(sip,sn , lip,lno,voted,ip,no) ≝
 1. [ sn ≥ lno ]       /* vote better */
 2.    Voting(sip,sn,voted,ip,no)
 3. + [ sn < lno ]       /* vote worse */
 4.    Voting(lip,lno,voted,ip,no)
```

to all destinations *dests*, and proceeds as P regardless of whether any of the transmissions is successful. The action **send**(*ms*) (synchronously) transmits a message to another process running on the same node. The sequential process **receive**(m).P receives any message m (a data value of type MSG) either from another node, from another sequential process running on the same node or from the client hooked up to the local node. It then proceeds as P, but with the data variable m bound to the value m. The submission of data from a client is modelled by the receipt of a message newpkt(d, *dip*), where the function newpkt generates a message containing the data d and the intended destination *dip*. Data is delivered to the client by **deliver**(*data*).

The layers of sequential and parallel processes usually define the behaviour of a protocol up to a point where it can be implemented; the other layers are used for reasoning, and include primitives for modelling dynamic topologies.

Processes 1 and 2, for example, describe a simple leader election protocol: each node in the network, which is assumed to be fully connected, holds a unique node identifier *ip* and a natural number n. Each node is initialised by (ξ,Voting(lip,lno,voted,ip,no)), with ξ(lip) = ξ(ip) = *ip*, ξ(lno) = ξ(no) = n, and ξ(voted) = false. The local variables lip and lno hold the identifier and the number of the current leader; the Boolean flag voted indicates whether the process partook in the election.

Process 1 allows the node to receive a ballot (message) B from another node (Lines 1 and 6, resp.). The message contains the sender's address ip, as well as its number no; these are stored in the local variables sip and sn. In case a message is received, the evaluation process Eval is called (Line 2). Once during the protocol (Line 3) each node can partake in the election and send its ballot, containing the node's own information (Line 4). After the message is sent, the flag voted is set to true (Line 5), and the node acts as if it had received this message.

Process 2 evaluates the information received. If the received number sn is greater than or equal to the number of the current leader lno, the current leader is set to sip and the current leader's number to sn, and the process returns to the main process; otherwise the information of the received message is disregarded.

When the protocol terminates—all nodes voted and all messages have been handled— all nodes have agreed on a leader, one holding the highest number no.

Once a model has been described in AWN, its behaviour is governed by the transitions allowed by the algebra's semantics. The formal semantics of AWN is given as structural operational semantics (sos) in the style of Plotkin [31] and describes how states evolve into another by performing *actions*. [11,12]

Table 2. Structural operational semantics (AWN) for **broadcast**

$$\xi, \mathbf{broadcast}(ms).P \xrightarrow{\mathbf{broadcast}(\xi(ms))} \xi, P \qquad \frac{P \xrightarrow{\mathbf{broadcast}(m)} P'}{ip : P : R \xrightarrow{R:\mathbf{*cast}(m)} ip : P' : R}$$

$$\frac{M \xrightarrow{R:\mathbf{*cast}(m)} M' \quad N \xrightarrow{H\neg K:\mathbf{arrive}(m)} N'}{M \| N \xrightarrow{R:\mathbf{*cast}(m)} M' \| N'} \begin{pmatrix} H \subseteq R \\ K \cap R = \emptyset \end{pmatrix} \qquad \frac{M \xrightarrow{R:\mathbf{*cast}(m)} M'}{[M] \xrightarrow{\tau} [M']}$$

Table 2 presents four sos-rules of AWN (out of 44), all describing behaviour w.r.t. broadcast. The first rule describes the behaviour of the sequential process $\mathbf{broadcast}(ms).P$, which performs the action $\mathbf{broadcast}(\xi(ms))$ without synchronisation. Here $\xi(ms)$ is the data value denoted by the expression ms when the variables occurring in ms are evaluated according to ξ. The second rule describes the **broadcast**-action on the node level: as the nodes in transmission range of node ip are known (stored in set R), this set is part of the new label and is used for synchronisation on the network layer. The third rule illustrates this partly. The action $R:\mathbf{*cast}(m)$ casts a message m that can be received by the set R of network nodes. AWN does not distinguish whether this message stems from a **broadcast-**, a **groupcast-** or a **unicast** action—the differences show up merely in the value of R. The action $H\neg K:\mathbf{arrive}(m)$ models that m simultaneously arrives at all addresses $ip \in H$, and fails to arrive at all addresses $ip \in K$. The third rule of Table 2 synchronises a $R:\mathbf{*cast}(m)$-action of one node with an $\mathbf{arrive}(m)$ of all other nodes. To finalise this synchronisation AWN features another two sos-rules: a symmetric form of the third rule, and a rule synchronising two $H\neg K:\mathbf{arrive}(m)$-actions. The side conditions ensure arrival of message m at all the nodes in the transmission range R of the $\mathbf{*cast}(m)$, and non-arrival at the other nodes. The fourth rule of Table 2 closes the network by the encapsulation operator $[_]$, and transforms the $R:\mathbf{*cast}(m)$-action into an internal action τ. The encapsulation guarantees that no messages will be received that have never been sent.

3 The Algebra mCRL2 and Its Associated Toolset

The milli Common Representation Language (mCRL2) [18] is a formal specification language with an associated toolset [8]. Similar to AWN, mCRL2 is a variant of standard process algebras with a formal semantics given as structural operational semantics in the style of Plotkin.

For our translation from AWN to mCRL2 we use only a fragment of mCRL2. In this section we briefly explain the syntax and semantics of those constructs of mCRL2 needed for our translation. As before, we can only show parts of the semantics, and refer to [18] for details.

Similar to AWN, mCRL2 comes with *defining equations*, called process equations in [18], having the form $X(\mathtt{d}_1 : \mathtt{D}_1, \ldots, \mathtt{d}_n : \mathtt{D}_n) \stackrel{\text{def}}{=} p$, where $\mathtt{d}_i$ are variables of sorts $\mathtt{D}_i$ and p a process expression defined by the following grammar.

Process 3 Voting in mCRL2

$\textbf{proc}\ \mathit{Voting}(\texttt{lip}{:}\mathrm{IP}, \texttt{lno}{:}\ \mathbb{N}, \texttt{voted}{:}\ \mathbb{B}, \texttt{ip}{:}\mathrm{IP}, \texttt{no}{:}\ \mathbb{N}) \stackrel{\text{def}}{=}$

1. $\sum_{\mathtt{D},\mathtt{D}'{:}\mathrm{Set(IP)},\mathtt{m}{:}\mathrm{MSG}} \textbf{receive}(\mathtt{D},\mathtt{D}',\mathtt{m})\cdot$
2. $\quad \sum_{\mathtt{sip}{:}\mathrm{IP},\mathtt{sn}{:}\mathbb{N}}((\mathtt{m} \approx \mathtt{MSG}(\mathtt{B},\mathtt{sip},\mathtt{sn})) \rightarrow (\mathbf{t}\cdot \mathit{Eval}(\mathtt{sip},\mathtt{sn},\mathtt{lip},\mathtt{lno},\mathtt{voted},\mathtt{ip},\mathtt{no})))$
3. $+\ (!\mathtt{voted}) \rightarrow \mathbf{t}\cdot($
4. $\quad \sum_{\mathtt{D}{:}\mathrm{Set(IP)}} \textbf{cast}(\mathrm{IP},\mathtt{D},\mathtt{MSG}(\mathtt{B},\mathtt{ip},\mathtt{no}))\cdot \mathit{Eval}(\mathtt{ip},\mathtt{sn},\mathtt{lip},\mathtt{lno},\mathtt{voted},\mathtt{ip},\mathtt{no})$
5. $\quad +\ \sum_{\mathtt{D},\mathtt{D}'{:}\mathrm{Set(IP)},\mathtt{m}{:}\mathrm{MSG}} \textbf{receive}(\mathtt{D},\mathtt{D}',\mathtt{m})\cdot$
6. $\quad\quad \sum_{\mathtt{sip}{:}\mathrm{IP},\mathtt{sn}{:}\mathbb{N}}((\mathtt{m} \approx \mathtt{MSG}(\mathtt{B},\mathtt{sip},\mathtt{sn})) \rightarrow (\mathbf{t}\cdot \mathit{Eval}(\mathtt{sip},\mathtt{sn},\mathtt{lip},\mathtt{lno},\mathtt{voted},\mathtt{ip},\mathtt{no}))))$

$$p ::= \alpha \mid p\cdot p \mid p + p \mid c \rightarrow p \mid \sum_{\mathtt{d}:\mathtt{D}} p \mid p \| p \mid X(\mathtt{u}_1, \ldots, \mathtt{u}_n) \mid \Gamma_C(p) \mid \nabla_V(p) \mid \rho_R(p) \mid \tau_I(p)$$

$$\alpha ::= \tau \mid a(\mathtt{u}_1, \ldots, \mathtt{u}_n) \mid \alpha|\alpha$$

Here α denotes a multi-action, c a Boolean, and the $\mathtt{u}_i$ are data expressions.

Actions form the basic building blocks of mCRL2. They consists of a name (taken from a given set) and some parameters, which are expressions denoting data values. Multi-actions are collections of actions that occur at the same time. A multi-action can be empty, denoted by τ; it is used as internal, non-observable action. $a(\vec{\mathtt{u}})$ denotes an action with name a and data parameters $\mathtt{u}_i$. Last, the multi-action $\alpha|\beta$ consists of the actions from both multi-actions α and β.

The process $p\cdot q$ behaves like p until p terminates, and then continues to behave as q. The process $p+q$ may act either as p or as q, depending on which of the processes can perform an action. If both are able to act, a non-deterministic choice is made. For a Boolean expression c, the process $c \rightarrow p$ acts like p if c evaluates to true, and deadlocks otherwise. The process $\sum_{\mathtt{d}:\mathtt{D}} p$ allows for a choice of p for any value d from $\mathtt{D}$ substituted for the variable $\mathtt{d}$—of course $\mathtt{d}$ can occur in p. The process $p\|q$ is a parallel composition of p and q. X is a process name, and $\mathtt{u}_i$ are data expressions of type $\mathtt{D}_i$, as declared in the defining equation; $X(\mathtt{u}_1, \ldots, \mathtt{u}_n)$ denotes a process call.

The communication operator $\Gamma_C(p)$ takes some actions out of a multi-action and replaces them with a single action, provided their data parts are equal. The set C describes the replacement by rules of the form $a_1|\cdots|a_1 \rightarrow c$. To enforce communication the allow operator $\nabla_V(p)$ only allows multi-actions listed in the set V to occur. The renaming operator $\rho_R(p)$ renames action names within p, where the set R lists rename rules of the form $a \rightarrow b$. Finally, the hiding operator $\tau_I(p)$ conceals all action names listed in I from the process p, replacing them by the internal action τ.

Process 3 models the same behaviour as Process 1, but written in mCRL2. In fact the presented specification has been translated by our tool (see Sect. 7); we have only changed minor issues such as variable names and line breaks to ease readability. An interesting issue when looking at the translation is that Process 3 features multiple sum-operators, where the AWN-specification shows none. While it may be understandable why the **receive**-actions (Lines 1 and 5) need to sum over all possible messages that could be received, the argument for sending messages (Line 4) is not straightforward. The reason is that we have to encode all possible transmission ranges $\mathtt{D}$; we elaborate on this in more detail in

Table 3. Structural operational semantics (mCRL2)

$$\alpha \xrightarrow{[\![\alpha]\!]} \checkmark \qquad \frac{p \xrightarrow{\omega} \checkmark}{p{\cdot}q \xrightarrow{\omega} q} \qquad \frac{p \xrightarrow{\omega} p'}{p+q \xrightarrow{\omega} p'} \qquad \frac{p \xrightarrow{\omega} p' \quad q \xrightarrow{\varpi} q'}{p\|q \xrightarrow{\omega|\varpi} p'\|q'}$$

$$\frac{p[\mathtt{d}{:=}t_e] \xrightarrow{\omega} p'}{\sum_{\mathtt{d:D}} p \xrightarrow{\omega} p'}\, e \in M_{\mathtt{D}} \qquad \frac{p \xrightarrow{\omega} p'}{c \rightarrow p \xrightarrow{\omega} p'}\, [\![c]\!] = \text{true} \qquad \frac{q[\mathtt{d}_1{:=}\mathtt{u}_1, \ldots, \mathtt{d}_n{:=}\mathtt{u}_n] \xrightarrow{\omega} q'}{X(\mathtt{u}_1, \ldots, \mathtt{u}_n) \xrightarrow{\omega} q'}$$

Sect. 5. Moreover, the AWN guard $[\mathtt{m} = \mathtt{B}(\mathtt{sip}, \mathtt{sn})]$ assigns values to $\mathtt{sip}$ and $\mathtt{sn}$ such that $\mathtt{m} = \mathtt{B}(\mathtt{sip}, \mathtt{sn})$; in mCRL2 this involves summing over all values $\mathtt{sip}$ and $\mathtt{sn}$ can take, in combination with the equality check $\mathtt{m} \approx \mathtt{MSG}(\mathtt{B}, \mathtt{sip}, \mathtt{sn})$.

Table 3 shows some rules of the structural operational semantics of mCRL2. Here $\checkmark$ indicates successful termination, and $[\![_]\!]$ is an interpretation function, sending syntactic expressions to semantic values. We have $[\![\tau]\!] = \tau$, $[\![a(\mathtt{u}_1, \ldots, \mathtt{u}_n)]\!] = a([\![\mathtt{u}_1]\!] \ldots, [\![\mathtt{u}_n]\!])$, and $[\![\alpha|\beta]\!] = [\![\alpha]\!]|[\![\beta]\!]$, where at the right-hand (semantic) side τ denotes the empty multiset, $[\![\alpha]\!]|[\![\beta]\!]$ the union of multisets $[\![\alpha]\!]$ and $[\![\beta]\!]$, and $a(e_1, \ldots, e_n)$ (the singleton multiset containing) the action a, whose parameters are now data values rather than data expressions. The first four rules are standard process algebra and (partly) characterise execution of an action, sequential composition (under successful termination), left choice and synchronisation, respectively. The first rule in the second line describes the sum operator. Here $M_{\mathtt{D}}$ is the set of data values of type $\mathtt{D}$ and t is a function—assumed to exist in mCRL2—that for each data value e returns a closed term t_e denoting e, i.e., $[\![t_e]\!] = e$. The second rule models a guard c; only if it evaluates to true, the process can proceed. The last rule of Table 3 defines recursion, where we assume a process $X(\mathtt{d}_1{:}\mathtt{D}_1 \ldots, \mathtt{d}_n{:}\mathtt{D}_n) \stackrel{\text{def}}{=} q$. mCRL2 also provides rules for the communication, the allow, and the restriction operator:

$$\frac{p \xrightarrow{\omega} p'}{\Gamma_C(p) \xrightarrow{\gamma_C(\omega)} \Gamma_C(p')} \qquad \frac{p \xrightarrow{\omega} p'}{\rho_R(p) \xrightarrow{R\bullet\omega} \rho_R(p')} \qquad \frac{p \xrightarrow{\omega} p'}{\nabla_V(p) \xrightarrow{\omega} \nabla_V(p')}\, \underline{\omega} \in V \cup \{\tau\}$$

Here the functions γ_C, and $R\bullet$ are the counterparts of Γ_C and ρ_R, resp., working on actions rather than processes. For example, $\gamma_{\{a|b \rightarrow c\}}(a|a|b|c) = a|c|c$. The stripped multi-action $\underline{\omega}$ is the result of removing all data from the multi-action ω.

Although mCRL2 works on top of an underlying data structure, it does not provide any syntactic construct for assignment.

mCRL2 comes with an associated toolset, consisting of about 50 different tools (see www.mcrl2.org). The toolset includes a user interface, which provides an easy way to read and analyse any mCRL2-process. Other tools help in manipulating and visualising state spaces, or provide support for automatic analysis. This includes classical model checking as well as checking properties by parameterised Boolean equation systems. The toolset also includes an interface allowing system analysis by the LTL/CTL/μ-calculus model checker LTSmin [23].

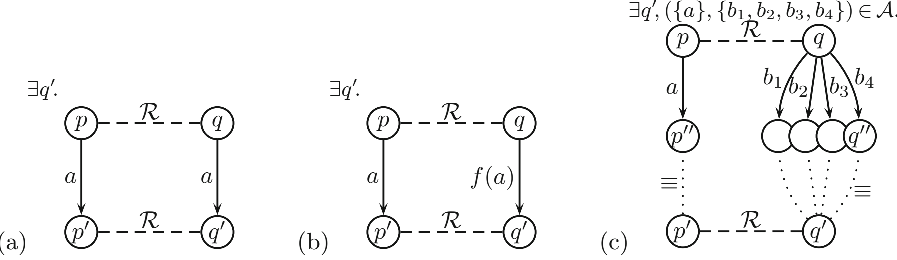


Fig. 1. Generalisations of simulations

4 Comparing Transition Systems

One goal of this paper is to translate a given specification written in AWN into an mCRL2-specification. Of course the generated specification should be related to the original one, so that we know which properties that should hold for the original specification can be checked in the translated specification.

The process algebras AWN and mCRL2 generate each a labelled transition system $(S, A, \rightarrow)$, where S is the set of all closed process algebra expressions, A is the set of possible actions, and $\rightarrow \subseteq S \times A \times S$ is the labelled transition relation where the transitions $P \xrightarrow{\alpha} Q$ are derived from the sos rules.

A standard technique to compare two transitions systems is (bi)simulation (e.g. [26]). A binary relation $\mathcal{R} \subseteq S_1 \times S_2$ is a *(strong) simulation*[1] [29] between transition systems $\mathcal{L}_1 = (S_1, A, \rightarrow_1)$ and $\mathcal{L}_2 = (S_2, A, \rightarrow_2)$ if it satisfies, for $a \in A$,

$$\text{if } p\,\mathcal{R}\,q \text{ and } p \xrightarrow{a}_1 p' \text{ then } \exists q'.\ q \xrightarrow{a}_2 q' \text{ and } p'\,\mathcal{R}\,q'\ .$$

Here $p \xrightarrow{a}_1 p'$ is a short-hand for $(p, a, p') \in \rightarrow_1$. A *bisimulation* is a symmetric simulation. If a bisimulation $\mathcal{R}$ with $p\,\mathcal{R}\,q$ exists then p and q are *bisimilar*.

Figure 1(a) illustrates the situation. Our definition slightly differs from the literature as it builds on two transition systems; the common definition presupposes $\mathcal{L}_1 = \mathcal{L}_2$. The definition requires an exact match of action labels. AWN and mCRL2 do not feature the same labels. For example, R:***cast**(m), which is an action label of AWN, does not follow the syntax of mCRL2-actions.

We relax the definition of simulation and say that $\mathcal{R} \subseteq S_1 \times S_2$ is a *simulation modulo renaming* between $\mathcal{L}_1 = (S_1, A_1, \rightarrow_1)$ and $\mathcal{L}_2 = (S_2, A_2, \rightarrow_2)$ for a bijective *renaming function* $f : A_1 \rightarrow A_2$ if it satisfies, for $a \in A_1$,

$$\text{if } p\,\mathcal{R}\,q \text{ and } p \xrightarrow{a}_1 p' \text{ then } \exists q'.\ q \xrightarrow{f(a)}_2 q' \text{ and } p'\,\mathcal{R}\,q'\ ;$$

see Fig. 1(b). A *bisimulation modulo renaming* is a symmetric simulation modulo renaming, using f and f^{-1}, respectively. Processes $p \in S_1$, $q \in S_2$ are *bisimilar modulo renaming* if a bisimulation modulo renaming $\mathcal{R}$ with $p\,\mathcal{R}\,q$ exists.

[1] This paper does not treat weak simulations, etc.; therefore we omit the word 'strong'.

It is well known that all safety properties are preserved under bisimilarity; and therefore also under bisimilarity modulo renaming, when the renaming function is applied to the safety property as well.[2]

Two mCRL2 processes p and q are *data congruent*, notation $p \equiv q$, if q can be obtained by replacing data expressions t occurring in p by expressions t' with $[\![t]\!] = [\![t']\!]$, i.e. evaluating to the same data value. For example $a(1{+}2) \equiv a(4{-}1)$. On AWN, we take $\equiv$ to be the identity. A (bi)simulation (modulo renaming) *up to* $\equiv$ is defined as above, but with $p' \equiv\mathcal{R}\equiv q'$ (using relational composition, denoted by juxtaposition) instead of $p' \,\mathcal{R}\, q'$. Using that $\equiv$ is a bisimulation [17], it follows from [26] that constructing a bisimulation $\mathcal{R}$ (modulo renaming) up to $\equiv$ with $p \,\mathcal{R}\, q$ suffices to show that p and q are bisimilar (modulo renaming).

In Sect. 5 we develop a translation between AWN and mCRL2, and we show that the translation is a bisimulation modulo renaming up to $\equiv$. However, this result only holds for encapsulated networks. When considering the other layers of AWN, a bisimulation cannot be established, not even modulo renaming. The reason is the layered design of AWN. While the set R of recipients of a **broadcast** is added only on the layer of node expressions, we need to introduce this set straightaway in mCRL2. On the process layer we do not have knowledge about nodes in transmission range. To include all possibilities, we require an entire collection of mCRL2-actions. We elaborate on this in the next section.

We call a relation $\mathcal{R} \subseteq S_1 \times S_2$ an $\mathcal{A}$-*warped simulation up to* $\equiv$ between transition systems $\mathcal{L}_1$ and $\mathcal{L}_2$ for a relation $\mathcal{A} \subseteq \mathcal{P}(A_1) \times \mathcal{P}(A_2)$ if it satisfies

$$\text{if } p \,\mathcal{R}\, q \text{ and } p \xrightarrow{a}_1 p'' \text{ then } \exists \mathrm{A}_1, \mathrm{A}_2, p', q'.$$
$$(a \in \mathrm{A}_1,\ p'' \equiv p',\ \mathrm{A}_1 \,\mathcal{A}\, \mathrm{A}_2,\ p \xrightarrow{\mathrm{A}_1}_1\equiv p',\ q \xrightarrow{\mathrm{A}_2}_2\equiv q' \text{ and } p' \,\mathcal{R}\, q'),$$

where $p \xrightarrow{\mathrm{A}}_1\equiv p' \Leftrightarrow_{df} \forall a \in \mathrm{A}.\exists p''.\, p \xrightarrow{a}_1 p'' \wedge p'' \equiv p'$. The definition requires a state q' such that all actions $a \in \mathrm{A}_2$ yield a transition to q', as illustrated in Figure 1(c).

An $\mathcal{A}$-*warped bisimulation up to* $\equiv$ is a symmetric $\mathcal{A}$-warped simulation up to $\equiv$, using $\mathcal{A}$ and $\mathcal{A}^{\smile} =_{df} \{(x,y) \mid (y,x) \in \mathcal{A}\}$, respectively.

Each (bi)simulation (up to $\equiv$) is also a (bi)simulation modulo renaming (up to $\equiv$)—using the identity as renaming; and each (bi)simulation modulo renaming up to $\equiv$ is an $\mathcal{A}$-warped (bi)simulation up to $\equiv$—with $\mathcal{A}{=}\{(\{a\},\{f(a)\}) | a{\in}A_1\}$.

5 From AWN to mCRL2

This section presents the formal translation from AWN-to mCRL2-processes.

Both process algebras are parameterised by the choice of an underlying data structure/abstract data type, and neither puts many restrictions on it; only the toolset associated to mCRL2 makes it more specific by predefining the most common concepts, such as integers, sets, lists, structs, etc. To ease readability, our presented translation assumes the same data structure underlying both process algebras. In particular, the translation maintains sorts—integers are mapped

[2] See [13] for a formal definition of safety property for labelled transition systems.

to integers etc. We also assume that variable names are the same. In the full version of this paper [17] we use translation functions that follow the detailed restrictions imposed on the respective data structures.

Tables 4–6 define the full translation, in recursive fashion.

Table 4. Translation function T (sequential processes)

$$
\begin{aligned}
\mathrm{T}_V(\zeta, \mathbf{broadcast}(ms).P) &= \textstyle\sum_{\mathtt{D}:\mathrm{Set(IP)}} \mathbf{cast}(\mathrm{IP}, \mathtt{D}, ms^\zeta) \cdot \mathrm{T}_V(\zeta, P)\\
\mathrm{T}_V(\zeta, \mathbf{groupcast}(dests, ms).P) &= \textstyle\sum_{\mathtt{D}:\mathrm{Set(IP)}} \mathbf{cast}(dests^\zeta, \mathtt{D}, ms^\zeta) \cdot \mathrm{T}_V(\zeta, P)\\
\mathrm{T}_V(\zeta, \mathbf{unicast}(dest, ms).P \blacktriangleright Q) &= \mathbf{cast}(\{dest^\zeta\}, \{dest^\zeta\}, ms^\zeta) \cdot \mathrm{T}_V(\zeta, P)\\
&\quad + \neg\mathbf{uni}(\{dest^\zeta\}, \emptyset, ms^\zeta) \cdot \mathrm{T}_V(\zeta, Q)\\
\mathrm{T}_V(\zeta, \mathbf{send}(ms).P) &= \mathbf{send}(\emptyset, \emptyset, ms^\zeta) \cdot \mathrm{T}_V(\zeta, P)\\
\mathrm{T}_V(\zeta, \mathbf{deliver}(data).P) &= \textstyle\sum_{\mathtt{ip}:\mathrm{IP}} \mathbf{del}(\mathtt{ip}, data^\zeta) \cdot \mathrm{T}_V(\zeta, P)\\
\mathrm{T}_V(\zeta, \mathbf{receive}(\mathtt{m}).P) &= \textstyle\sum_{\substack{\mathtt{D},\mathtt{D}':\mathrm{Set(IP)}\\ \mathtt{m}:\mathrm{MSG}}} \mathbf{receive}(\mathtt{D}, \mathtt{D}', \mathtt{m}) \cdot \mathrm{T}_{V\cup\{\mathtt{m}\}}(\zeta^{\setminus \mathtt{m}}, P)\\
\mathrm{T}_V(\zeta, [\![\mathtt{v} := exp]\!]\ P) &= \textstyle\sum_{\mathtt{y}:\mathrm{sort}(\mathtt{v})} (\mathtt{y} = exp^\zeta) \rightarrow\\
&\quad (\textstyle\sum_{\mathtt{v}:\mathrm{sort}(\mathtt{y})} (\mathtt{v} = \mathtt{y}) \rightarrow \mathbf{t} \cdot \mathrm{T}_{V\cup\{\mathtt{v}\}}(\zeta^{\setminus \mathtt{v}}, P))\\
\mathrm{T}_V(\zeta, X(exp_1, \cdots, exp_n)) &= X(exp_1^\zeta, \cdots, exp_n^\zeta)\\
\mathrm{T}_V(\zeta, P + Q) &= \mathrm{T}_V(\zeta, P) + \mathrm{T}_V(\zeta, Q)\\
\mathrm{T}_V(\zeta, [\varphi]P) &= \textstyle\sum_{\mathrm{FV}(\varphi)\setminus V} \varphi^\zeta \rightarrow \mathbf{t} \cdot \mathrm{T}_{V\cup\mathrm{FV}(\varphi)}(\zeta, P)
\end{aligned}
$$

Table 4 lists the translation rules for sequential processes. On this level, our translation function operates on sequential process expressions P and additionally carries two parameters: the set V of data variables maintained by P, and a valuation ζ of *some* of these variables. So $\mathrm{DOM}(\zeta) \subseteq V$. ζ evaluates all variables that in the translation to mCRL2 are turned into constants, or other closed data expressions; the variables in $V \setminus \mathrm{DOM}(\zeta)$ remain variables upon translation. exp^ζ denotes the mCRL2-expression *exp* with $t_{\zeta(x)}$ substituted for each $x \in \mathrm{DOM}(\zeta)$. The set V is always the domain of the valuation ξ of a sequential process (ξ, P); hence the ζ used as a parameter in the translation is only a part of ξ.

The first two equations translate **broadcast** and **groupcast**-actions in a similar fashion. Since mCRL2 does not allow to alter the number nor the type of arguments of an action, we have to add all parameters from the beginning. As a consequence the action **cast** carries three arguments: the intended destinations of a message (a set of addresses), the actual destinations, and the message itself. For **broadcast** the set of intended addresses is the set of all IP addresses; for **groupcast** this set is determined by the AWN-primitive. The second argument hinges on the set of reachable destinations (destinations in transmission range), which is only specified on the level of node expressions—see e.g. Rule 2 of Table 2. To allow arbitrary sets of destinations these rules use the sum operator of mCRL2 ($\sum$)—the correct set of destinations is chosen later, by using the parallel operator $\|$. For the translation we have to assume that $\mathtt{D}$ and $\mathtt{D}'$ are fresh variables; in [17] we list all required side conditions, which we skip here to

Table 5. Translation function T (defining equation and parallel processes)

$$\mathrm{T}(X(\mathtt{v}_1, \cdots, \mathtt{v}_n) \stackrel{def}{=} P) = \big(X(\mathtt{v}_1{:}\mathrm{sort}(\mathtt{v}_1), \cdots, \mathtt{v}_n{:}\mathrm{sort}(\mathtt{v}_n)) \stackrel{\mathrm{def}}{=} \mathrm{T}_{\{\mathtt{v}_1,\cdots,\mathtt{v}_n\}}(\emptyset, P)\big)$$

$$\mathrm{T}((\xi, P)) = \mathrm{T}_{\mathrm{DOM}(\xi)}(\xi, P)$$

$$\mathrm{T}(P \langle\!\langle Q) = \nabla_V \Gamma_{\{\mathbf{r}|\mathbf{s}\to\mathbf{t}\}}(\rho_{\{\mathbf{receive}\to\mathbf{r}\}}\mathrm{T}(P) \| \rho_{\{\mathbf{send}\to\mathbf{s}\}}\mathrm{T}(Q))$$
$$\text{where } V = \{\mathbf{t}, \mathbf{cast}, \neg\mathbf{uni}, \mathbf{send}, \mathbf{del}, \mathbf{receive}\}$$

ease readability. After the **broadcast**-action has been translated, the remaining process P is handled by the same translation function. The **unicast** primitive uses a similar translation in case of successful transmission, but also allows the possibility of failure, which is handled by the action $\neg$**uni**.

The translation of the **send**-primitive is straightforward; the only subtlety is that the translation has to have as many arguments as the **cast**-action, since both synchronise with **receive**—we use the empty set $\emptyset$ as dummy parameter. The **deliver**-action delivers *data* to the client; as this can happen at any network node, we sum over all possible recipients `ip`. The translation of **receive** follows the style of **broadcast** and **groupcast**, and synchronises with the **cast**-action later on. Hence it needs the same number of arguments as that action; as all parameters are unknown, we sum over all of them. After the **receive**-action, the variable `m` is added to the set V of variables maintained by the AWN-process P. However, since in the mCRL2 translation it occurs under the scope of a sum operator, it is not instantiated with a concrete message in the translation of P, and hence is removed from the domain of ζ—notation $\zeta^{\setminus\mathtt{m}}$.

Since mCRL2 does not provide a primitive for assignment, the translation of $[\![\mathtt{v} := exp]\!]P$ is non-trivial. The idea behind our translation is to sum over all possible values of `v`, and use a guard to pick the right value. A first rendering of the translation rule would be $\sum_{\mathtt{v}:\mathrm{sort}(\mathtt{v})}(\mathtt{v} = exp^{\zeta}) \to X$, where X is a process to be determined. This sum-guard combination works for many cases; it fails when the expression contains the variable itself. An example is the increment of a variable: $[\![\mathtt{x} = \mathtt{x}{+}1]\!]$. To resolve this problem we use a standard technique of programming and introduce a fresh variable `y`. We then split the assignment and calculate $[\![\mathtt{y} = \mathtt{x}{+}1]\!][\![\mathtt{x} = \mathtt{y}]\!]$. Both assignments are transformed into sum-guard form. Since we aim at strong bisimilarity and the assignment rule of AWN produces a silent action τ, we do something similar for mCRL2. For technical reasons[3] we cannot use a τ-action, and use an action named **t** instead.

Both AWN and mCRL2 feature process calls and an operator for (binary) choice with the same semantics; their obvious translation is given by the next two lines of Table 4. The guard of AWN translates to a guard in mCRL2. However, AWN assigns variables that occur free in φ and that are not maintained by the current process in a non-deterministic manner such that φ evaluates to true.

[3] Using that $\tau|\tau = \tau$, the fourth rule of Table 3 allows any two parallel τ-transitions in mCRL2 to synchronise, which is not possible in AWN. For this reason, τ-actions in AWN are translated in an action **t** of mCRL2, which is turned into a τ only at the outermost layer, where no further parallel compositions are encountered.

Table 6. Translation function T (network nodes and networks)

$$\mathrm{T}(ip : P : R) = \nabla_V \Gamma_C(\mathrm{T}(P) \| G(ip, R))$$

where $V = \{\mathbf{t}, \mathbf{starcast}, \mathbf{arrive}, \mathbf{deliver}, \mathbf{connect}, \mathbf{disconnect}\}$

where $C = \{\mathbf{cast}|\overline{\mathbf{cast}} \to \mathbf{starcast}, \neg\mathbf{uni}|\overline{\neg\mathbf{uni}} \to \mathbf{t}, \mathbf{del}|\overline{\mathbf{del}} \to \mathbf{deliver}, \mathbf{receive}|\overline{\mathbf{receive}} \to \mathbf{arrive}\}$

$$\begin{aligned}
\text{where } G(\mathtt{ip}, \mathtt{R}) \stackrel{\text{def}}{=} & \textstyle\sum_{\substack{\mathtt{D},\mathtt{D}':\mathrm{Set(IP)}\\ \mathtt{m}:\mathrm{MSG}}} (\mathtt{R} \cap \mathtt{D} = \mathtt{D}') \to \overline{\mathbf{cast}}(\mathtt{D}, \mathtt{D}', \mathtt{m}) \cdot G(\mathtt{ip}, \mathtt{R}) \\
& + \textstyle\sum_{\substack{\mathtt{d}:\mathrm{IP}\\ \mathtt{m}:\mathrm{MSG}}} (\mathtt{d} \notin \mathtt{R}) \to \overline{\neg\mathbf{uni}}(\{\mathtt{d}\}, \emptyset, \mathtt{m}) \cdot G(\mathtt{ip}, \mathtt{R}) \\
& + \textstyle\sum_{\mathtt{data}:\mathrm{DATA}} \overline{\mathbf{del}}(\mathtt{ip}, \mathtt{data}) \cdot G(\mathtt{ip}, \mathtt{R}) \\
& + \textstyle\sum_{\mathtt{ip}':\mathrm{IP}} \mathbf{connect}(\mathtt{ip}, \mathtt{ip}') \cdot G(\mathtt{ip}, \mathtt{R} \cup \{\mathtt{ip}'\}) \\
& + \textstyle\sum_{\mathtt{ip}':\mathrm{IP}} \mathbf{connect}(\mathtt{ip}', \mathtt{ip}) \cdot G(\mathtt{ip}, \mathtt{R} \cup \{\mathtt{ip}'\}) \\
& + \textstyle\sum_{\mathtt{ip}',\mathtt{ip}'':\mathrm{IP}} (\mathtt{ip} \notin \{\mathtt{ip}', \mathtt{ip}''\}) \to \mathbf{connect}(\mathtt{ip}', \mathtt{ip}'') \cdot G(\mathtt{ip}, \mathtt{R}) \\
& + \textstyle\sum_{\mathtt{ip}':\mathrm{IP}} \mathbf{disconnect}(\mathtt{ip}, \mathtt{ip}') \cdot G(\mathtt{ip}, \mathtt{R} \backslash \{\mathtt{ip}'\}) \\
& + \textstyle\sum_{\mathtt{ip}':\mathrm{IP}} \mathbf{disconnect}(\mathtt{ip}', \mathtt{ip}) \cdot G(\mathtt{ip}, \mathtt{R} \backslash \{\mathtt{ip}'\}) \\
& + \textstyle\sum_{\mathtt{ip}',\mathtt{ip}'':\mathrm{IP}} (\mathtt{ip} \notin \{\mathtt{ip}', \mathtt{ip}''\}) \to \mathbf{disconnect}(\mathtt{ip}', \mathtt{ip}'') \cdot G(\mathtt{ip}, \mathtt{R}) \\
& + \textstyle\sum_{\substack{\mathtt{D},\mathtt{D}':\mathrm{Set(IP)}\\ \mathtt{m}:\mathrm{MSG}}} (\mathtt{ip} \in \mathtt{D}') \to \overline{\mathbf{receive}}(\mathtt{D}, \mathtt{D}', \mathtt{m}) \cdot G(\mathtt{ip}, \mathtt{R}) \\
& + \textstyle\sum_{\substack{\mathtt{D},\mathtt{D}':\mathrm{Set(IP)}\\ \mathtt{m}:\mathrm{MSG}}} (\mathtt{ip} \notin \mathtt{D}') \to \mathbf{arrive}(\mathtt{D}, \mathtt{D}', \mathtt{m}) \cdot G(\mathtt{ip}, \mathtt{R})
\end{aligned}$$

$$\mathrm{T}(M \| N) = \rho_R \nabla_V \Gamma_{\{\mathbf{arrive}|\mathbf{arrive} \to \mathbf{a}\}} \Gamma_C(\mathrm{T}(M) \| \mathrm{T}(N))$$

where $R = \{\mathbf{a} \to \mathbf{arrive}, \mathbf{c} \to \mathbf{connect}, \mathbf{d} \to \mathbf{disconnect}, \mathbf{s} \to \mathbf{starcast}\}$

where $V = \{\mathbf{a}, \mathbf{c}, \mathbf{d}, \mathbf{deliver}, \mathbf{s}, \mathbf{t}\}$

where $C = \{\mathbf{starcast}|\mathbf{arrive} \to \mathbf{s}, \mathbf{connect}|\mathbf{connect} \to \mathbf{c}, \mathbf{disconnect}|\mathbf{disconnect} \to \mathbf{d}\}$

$$\mathrm{T}([M]) = \tau_{\{\mathbf{t}\}} \nabla_V \rho_{\{\mathbf{starcast} \to \mathbf{t}\}} \Gamma_C(\mathrm{T}(M) \| H)$$

where $V = \{\mathbf{t}, \mathbf{newpkt}, \mathbf{deliver}, \mathbf{connect}, \mathbf{disconnect}\}$

where $C = \{\overline{\mathbf{newpkt}}|\mathbf{arrive} \to \mathbf{newpkt}\}$

where $H \stackrel{\text{def}}{=} \sum_{\mathtt{ip}:\mathrm{IP}, \mathtt{data}:\mathrm{DATA}, \mathtt{dest}:\mathrm{IP}} \overline{\mathbf{newpkt}}(\{\mathtt{ip}\}, \{\mathtt{ip}\}, \mathtt{newpkt}(\mathtt{data}, \mathtt{dest})) \cdot H$

We model the same behaviour by a sum over those variables that can be chosen freely; here the set $\mathrm{FV}(\varphi)$ contains all free variables of the Boolean formula φ. This is the only place in the translation where the parameter V is used at all. The mCRL2-expression of this rule simplifies to $\varphi^\zeta \to \mathbf{t} \cdot \mathrm{T}_V(\zeta, P)$ in case all free variables of φ occur in V.

Table 5 first presents the translation of defining equations, which is straightforward. The set V of variables maintained by P consists of the parameters $\mathtt{v}_i$ of the process name X. The table also lists the translation rules for parallel processes. The rule for (ξ, P) merely needs to initialise the set V as $\mathrm{DOM}(\xi)$. The last rule handles the (asymmetric) parallel operator of parallel processes. This operator allows and enforces synchronisation of a **send**-action on the right with a **receive**-action on the left only. For example, in the expression $(P \langle\!\langle Q) \langle\!\langle R$ the **send** and **receive**-actions of Q can communicate only with P and R, respectively, but the **receive**-actions of R, as well as the **send**-actions of P, remain available for communication with the environment. Since mCRL2 only offers

Table 7. Action relation $\mathcal{A}$

$$\mathcal{A} =_{df} \Big\{ (\{\tau\},\{\mathbf{t}\}),\ (\{\mathbf{broadcast}(\xi(ms))\},\{\mathbf{cast}([\![\mathrm{IP}]\!],[\![\hat{\mathtt{D}}]\!],[\![\xi(ms)]\!]) \mid \hat{\mathtt{D}}{:}\mathrm{Set(IP)}\}),$$
$$(\{\mathbf{groupcast}(\xi(dests),\xi(ms))\},\{\mathbf{cast}([\![\xi(dests)]\!],[\![\hat{\mathtt{D}}]\!],[\![\xi(ms)]\!]) \mid \hat{\mathtt{D}}{:}\mathrm{Set(IP)}\}),$$
$$(\{\mathbf{unicast}(\xi(dest),\xi(ms))\},\{\mathbf{cast}([\![\xi(\{dest\})]\!],[\![\xi(\{dest\})]\!],[\![\xi(ms)]\!])\}),$$
$$(\{\neg\mathbf{unicast}(\xi(dest),\xi(ms))\},\{\neg\mathbf{uni}([\![\xi(\{dest\})]\!],[\![\emptyset]\!],[\![\xi(ms)]\!])\}),$$
$$(\{\mathbf{send}(\xi(ms))\},\{\mathbf{send}([\![\emptyset]\!],[\![\emptyset]\!],[\![\xi(ms)]\!])\}),$$
$$(\{\mathbf{deliver}(\xi(data))\},\{\mathbf{del}([\![\hat{\mathtt{ip}}]\!],[\![\xi(data)]\!]) \mid \hat{\mathtt{ip}}{:}\mathrm{IP}\}),$$
$$(\{\mathbf{receive}(m)\},\{\mathbf{receive}([\![\hat{\mathtt{D}}]\!],[\![\hat{\mathtt{D}}']\!],m) \mid \hat{\mathtt{D}},\hat{\mathtt{D}}'{:}\mathrm{Set(IP)}\}),$$
$$(\{R{:}\mathbf{*cast}(m)\},\{\mathbf{starcast}(D,R,m)\}),\ (\{ip{:}\mathbf{deliver}(d)\},\{\mathbf{deliver}(ip,d)\}),$$
$$(\{H\neg K{:}\mathbf{arrive}(m)\},\{\mathbf{arrive}([\![\hat{\mathtt{D}}]\!],[\![\hat{\mathtt{D}}']\!],m) \mid {}^{\hat{\mathtt{D}},\hat{\mathtt{D}}'{:}\mathrm{Set(IP)},\ \hat{\mathtt{D}}'\subseteq\hat{\mathtt{D}}}_{H\subseteq\hat{\mathtt{D}}',\ K\cap\hat{\mathtt{D}}'=\emptyset}\}),$$
$$(\{\mathbf{connect}(ip',ip'')\},\{\mathbf{connect}(ip',ip'')\}),$$
$$(\{\mathbf{disconnect}(ip',ip'')\},\{\mathbf{disconnect}(ip',ip'')\}),$$
$$(\{ip{:}\mathbf{newpkt}(d,dip)\},\{\mathbf{newpkt}(\{ip\},\{ip\},\mathtt{newpkt}(d,dip))\}),(\{\tau\},\{\tau\})$$
$$\Big|\ m,\xi(ms){:}\mathrm{MSG};\ ip,ip',ip'',dip,dest{:}\mathrm{IP};\ d,\xi(data){:}\mathrm{DATA};$$
$$dests,R,D,H,K{:}\mathrm{Set(IP)};\ R\subseteq D\Big\}$$

a standard, symmetric parallel operator, we model the behaviour by combining renaming, communication and allow operators. By renaming **receive** to **r** in the left process and **send** to **s** in the right process we guarantee synchronisation of the corresponding actions; the communication operator renames the synchronised action into **t**, which later becomes an internal action τ. To enforce synchronisation, we apply the allow-operator ∇, and restrict the set of actions to those possible. Among others this disallows all proper multi-actions.

Table 6 shows the translation rules for network nodes, networks and encapsulated networks. All rules use combinations of the mCRL2-operators ∇ and Γ, similar to the last rule of Table 5. The process G is used to select the correct set of nodes receiving a message—remember that we sum over all possible sets on the level of sequential processes (see Table 4). It also introduces the primitives for changing network topologies, such as **connect**ing and **disconnect**ing two nodes. The rule for $\|$ features two Γ-operators, as mCRL2 forbids a single one to have overlapping redexes. The encapsulation allows only actions with name **newpkt**, **deliver**, **connect**, **disconnect**, as well as the 'to-be' silent action **t**. The process H handles the injection of a new data packet, where all parameters (point of injection `ip`, the destination `dest` as well the content `data` of the message) are unknown; we sum over these values.

This concludes the formal definition and explanation of the translation from the process algebra AWN into the process algebra mCRL2.

6 Correctness of the Translation

This section describes the relationship between AWN-specifications and their counterparts in mCRL2. We establish that our translation forms a warped bisimulation up to $\equiv$ on all layers of AWN; and a bisimulation modulo renaming up to $\equiv$ for encapsulated networks.

Theorem 6.1. The relation $\{(P, \mathrm{T}(P)) \mid P \text{ is an AWN-process}\}$ is an $\mathcal{A}$-warped simulation up to $\equiv$, where $\mathcal{A}$ is the action relation of Table 7.

Proof Sketch. We need to show that

$$\text{if } P \xrightarrow{a} P' \text{ then } \exists \mathrm{A}_1, \mathrm{A}_2.\ (P \xrightarrow{\mathrm{A}_1} P',\ \mathrm{T}(P) \xrightarrow{\mathrm{A}_2}\equiv \mathrm{T}(P'),\ \mathrm{A}_1\,\mathcal{A}\,\mathrm{A}_2 \text{ and } a \in \mathrm{A}_1),$$

for all AWN action labels a. We prove this implication by structural induction on the derivation of $P \xrightarrow{a} P'$ from the inference rules of AWN.

The base cases consider all sos-rules of AWN without premises, such as the first rule of Table 2. Out of the 14 bases cases we only present the proof for this rule, and prove that there are sets A_1 and A_2 satisfying the above properties.

Since $\mathbf{broadcast}(\xi(ms)) \in \mathrm{A}_1$, Table 7 implies $\mathrm{A}_1{=}\mathbf{broadcast}(\xi(ms))$ and $\mathrm{A}_2{=}\{\mathbf{cast}([\![\xi(dests)]\!], [\![\hat{\mathtt{D}}]\!], [\![\xi(ms)]\!]) \mid \hat{\mathtt{D}}{:}\mathrm{Set}(\mathrm{IP})\}$. It suffices to find a derivation in mCRL2 such that $\mathrm{T}(\xi, \mathbf{broadcast}(ms).p) \xrightarrow{a} \mathrm{T}(\xi, p)$, for all $a \in \mathrm{A}_2$. For arbitrary $\hat{\mathtt{D}}$ we have

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\overline{\mathbf{cast}(\mathrm{IP}, \hat{\mathtt{D}}, \xi(ms)) \xrightarrow{[\![\mathbf{cast}(\mathrm{IP},\hat{\mathtt{D}},\xi(ms))]\!]} \checkmark}}{\mathbf{cast}(\mathrm{IP}, \hat{\mathtt{D}}, \xi(ms))\cdot \mathrm{T}_{\mathrm{DOM}(\xi)}(\xi, P) \xrightarrow{\mathbf{cast}([\![\mathrm{IP}]\!],[\![\hat{\mathtt{D}}]\!],[\![\xi(ms)]\!])} \mathrm{T}_{\mathrm{DOM}(\xi)}(\xi, P)}}{\big(\mathbf{cast}(\mathrm{IP}, \mathtt{D}, \xi(ms))\cdot \mathrm{T}_{\mathrm{DOM}(\xi)}(\xi, P)\big)[\mathtt{D} := \hat{\mathtt{D}}] \xrightarrow{\mathbf{cast}([\![\mathrm{IP}]\!],[\![\hat{\mathtt{D}}]\!],[\![\xi(ms)]\!])} \mathrm{T}_{\mathrm{DOM}(\xi)}(\xi, P)}}{\sum_{\mathtt{D}:\mathrm{Set}(\mathrm{IP})} \mathbf{cast}(\mathrm{IP}, \mathtt{D}, \xi(ms))\cdot \mathrm{T}_{\mathrm{DOM}(\xi)}(\xi, P) \xrightarrow{\mathbf{cast}([\![\mathrm{IP}]\!],[\![\hat{\mathtt{D}}]\!],[\![\xi(ms)]\!])} \mathrm{T}_{\mathrm{DOM}(\xi)}(\xi, P)}}{\mathrm{T}_{\mathrm{DOM}(\xi)}(\xi, \mathbf{broadcast}(ms).P) \xrightarrow{\mathbf{cast}([\![\mathrm{IP}]\!],[\![\hat{\mathtt{D}}]\!],[\![\xi(ms)]\!])} \mathrm{T}_{\mathrm{DOM}(\xi)}(\xi, P)}}{\mathrm{T}(\xi, \mathbf{broadcast}(ms).P) \xrightarrow{\mathbf{cast}([\![\mathrm{IP}]\!],[\![\hat{\mathtt{D}}]\!],[\![\xi(ms)]\!])} \mathrm{T}(\xi, P)}$$

The validity of the first transition follows from the first rule of Table 3. The second one follows from the second rule, and a distributivity property of the interpretation function $[\![_]\!]$ (see Sect. 3). To use the sos-rule for sum of Table 3 in Step 4, we rewrite the process on the left-hand side using substitution. The remaining two steps use the presented translation function (Line 1 of Table 4 and Line 2 of Table 5).

The induction step covers all rules that have at least one premise. Out of the 30 cases we present only the proof of

$$\frac{P \xrightarrow{\mathbf{broadcast}(m)} P'}{ip : P : R \xrightarrow{R:\mathbf{*cast}(m)} ip : P' : R}$$

Table 7 allows $\mathrm{A}_1 = \{R : \mathbf{*cast}(m)\}$ and $\mathrm{A}_2 = \{\mathbf{starcast}(\mathrm{IP}, R, m)\}$, choosing $D = \mathrm{IP}$. The induction step is proven by providing a derivation in mCRL2 for $\mathrm{T}(ip : P : R) \xrightarrow{a} \mathrm{T}(ip : P' : R)$, for all $a \in A_2$. First, we analyse the process G.

$$\begin{array}{c}
\hline
\overline{\mathbf{cast}}(\mathrm{IP}, R, t_m) \xrightarrow{[\![\overline{\mathbf{cast}}(\mathrm{IP},R,t_m)]\!]} \checkmark \\
\hline
\overline{\mathbf{cast}}(\mathrm{IP}, R, t_m)\cdot G(ip, R) \xrightarrow{\overline{\mathbf{cast}}(\mathrm{IP},R,m)} G(ip, R) \\
\hline
(R \cap \mathrm{IP} = R) \to \overline{\mathbf{cast}}(\mathrm{IP}, R, t_m)\cdot G(ip, R) \xrightarrow{\overline{\mathbf{cast}}(\mathrm{IP},R,m)} G(ip, R) \\
\hline
(c' \to \overline{\mathbf{cast}}(\mathtt{D}, \mathtt{D}', \mathtt{m})\cdot G(ip, R))[\mathtt{D}{:=}\mathrm{IP}, \mathtt{D}'{:=}R, \mathtt{m}{:=}t_m] \xrightarrow{\overline{\mathbf{cast}}(\mathrm{IP},R,m)} G(ip, R) \\
\hline
\sum_{\substack{\mathtt{D},\mathtt{D}':\mathrm{Set(IP)} \\ \mathtt{m}:\mathrm{MSG}}} c' \to \overline{\mathbf{cast}}(\mathtt{D}, \mathtt{D}', \mathtt{m})\cdot G(ip, R) \xrightarrow{\overline{\mathbf{cast}}(\mathrm{IP},R,m)} G(ip, R) \\
\hline
\sum_{\substack{\mathtt{D},\mathtt{D}':\mathrm{Set(IP)} \\ \mathtt{m}:\mathrm{MSG}}} c' \to \overline{\mathbf{cast}}(\mathtt{D}, \mathtt{D}', \mathtt{m})\cdot G(ip, R) + S' \xrightarrow{\overline{\mathbf{cast}}(\mathrm{IP},R,m)} G(ip, R) \\
\hline
(\sum_{\substack{\mathtt{D},\mathtt{D}':\mathrm{Set(IP)} \\ \mathtt{m}:\mathrm{MSG}}} c \to \overline{\mathbf{cast}}(\mathtt{D}, \mathtt{D}', \mathtt{m})\cdot G(\mathtt{ip}, \mathtt{R}) + S)[\mathtt{ip}{:=}ip, \mathtt{R}{:=}R] \xrightarrow{\overline{\mathbf{cast}}(\mathrm{IP},R,m)} G(ip, R) \\
\hline
G(ip, R) \xrightarrow{\overline{\mathbf{cast}}(\mathrm{IP},R,m)} G(ip, R)
\end{array}$$

where S is an expression equal to all summands of G except the first one, and $S' = S[\mathtt{ip}{:=}ip, \mathtt{R}{:=}R]$. Moreover, $c = (\mathtt{R} \cap \mathtt{D} = \mathtt{D}')$ and $c' = (R \cap \mathtt{D} = \mathtt{D}')$. We use '$R$' and 'IP' as data values as well as expressions denoting these, so $[\![R]\!] = R$ and $[\![\mathrm{IP}]\!] = \mathrm{IP}$.

As for the previous derivation the first two steps follow from the first two rules of Table 3, using $[\![t_m]\!] = m$. The following step applies the rule for guard (sixth rule in Table 3), using $[\![R \cap \mathrm{IP} = R]\!] = \mathrm{true}$ as side condition. As before we use substitution such that we can apply the sum operator. We then use the rule of binary choice (third one in Table 3) and substitution again. The final step applies the recursion rule of mCRL2 (last one in the table).

Since there is only one pair $(\mathrm{B}_1, \mathrm{B}_2) \in \mathcal{A}$ with $\mathbf{broadcast}(m) \in \mathrm{B}_1$ (see Table 7), $\mathrm{T}(P) \xrightarrow{\mathbf{cast}(\mathrm{IP},R,m)} \mathrm{T}(P')$, using the induction hypothesis. We combine this fact with the conclusion of the derivation above.

$$\begin{array}{c}
\dfrac{}{\mathrm{T}(P) \xrightarrow{\mathbf{cast}(\mathrm{IP},R,m)} \mathrm{T}(P')}\ \text{Induction hypothesis} \qquad \dfrac{}{G(ip, R) \xrightarrow{\overline{\mathbf{cast}}(\mathrm{IP},R,m)} G(ip, R)} \\
\hline
\mathrm{T}(P) \| G(ip, R) \xrightarrow{\mathbf{cast}(\mathrm{IP},R,m)|\overline{\mathbf{cast}}(\mathrm{IP},R,m)} \mathrm{T}(P') \| G(ip, R) \\
\hline
\Gamma_C(\mathrm{T}(P) \| G(ip, R)) \xrightarrow{\gamma_C(\mathbf{cast}(\mathrm{IP},R,m)|\overline{\mathbf{cast}}(\mathrm{IP},R,m))} \Gamma_C(\mathrm{T}(P') \| G(ip, R)) \\
\hline
\Gamma_C(\mathrm{T}(P) \| G(ip, R)) \xrightarrow{\mathbf{starcast}(\mathrm{IP},R,m)} \Gamma_C(\mathrm{T}(P') \| G(ip, R)) \\
\hline
\nabla_V \Gamma_C(\mathrm{T}(P) \| G(ip, R)) \xrightarrow{\mathbf{starcast}(\mathrm{IP},R,m)} \nabla_V \Gamma_C(\mathrm{T}(P') \| G(ip, R)) \\
\hline
\mathrm{T}(ip : P : R) \xrightarrow{\mathbf{starcast}(\mathrm{IP},R,m)} \mathrm{T}(ip : P' : R)
\end{array}$$

The derivation is straightforward, using the synchronisation rule of Table 3 and the rules for mCRL2-operators listed on Page 7. This finishes the induction step for the **broadcast**-rule. □

A full and detailed proof can be found in [17].

We have shown that translated processes simulate original processes. We now turn to the opposite direction.

Theorem 6.2. The relation $\{(\mathrm{T}(P), P) \mid P \text{ is an AWN-process}\}$ is an $\mathcal{A}^{\smile}$-warped simulation up to $\equiv$, where $\mathcal{A}^{\smile}$ is the converse action relation of Table 7.

Similar to Theorem 6.1, the proof is by structural induction. In contrast to the above proof, the proof of Theorem 6.2 is more complicated. The reason is that the relation $\mathcal{A}$ is a function, whereas $\mathcal{A}^{\smile}$ is not. As a consequence the individual cases (base cases and induction steps) contain several case distinctions. For example, an action labelled $\mathbf{cast}([\![\mathtt{D}]\!], [\![\mathtt{D'}]\!], [\![\mathtt{m}]\!])$ could stem from a **broadcast**, a **groupcast** or a **unicast**-action in AWN. The action labelled **t** is even worse: it can stem from an internal action τ, the action **starcast**, from a synchronisation $\mathbf{uni}|\neg\mathbf{uni}$, etc. Again, the full proof can be found in [17].

Corollary 6.3. The relation $\{(P, \mathrm{T}(P)) \mid P$ is an AWN-process$\}$ is an $\mathcal{A}^{\smile}$-warped bisimulation up to $\equiv$.

Using this result we are now ready to prove the main theorem.

Theorem 6.4. The relation $\{(P, \mathrm{T}(P)) \mid P$ is an encapsulated network expression in AWN$\}$[4] is a bisimulation modulo renaming up to $\equiv$ w.r.t. to the bijective renaming function f, defined as

$$\mathrm{f}(a) =_{df} \begin{cases} \tau & \text{if } a = \tau \\ \mathbf{deliver}(\mathit{ip}, d) & \text{if } a = \mathit{ip}\,{:}\,\mathbf{deliver}(d) \\ \mathbf{connect}(\mathit{ip}', \mathit{ip}'') & \text{if } a = \mathbf{connect}(\mathit{ip}', \mathit{ip}'') \\ \mathbf{disconnect}(\mathit{ip}', \mathit{ip}'') & \text{if } a = \mathbf{disconnect}(\mathit{ip}', \mathit{ip}'') \\ \mathbf{newpkt}(\{\mathit{ip}\}, \{\mathit{ip}\}, \mathtt{newpkt}(d, \mathit{dip})) & \text{if } a = \mathit{ip}\,{:}\,\mathbf{newpkt}(d, \mathit{dip}) \,. \end{cases}$$

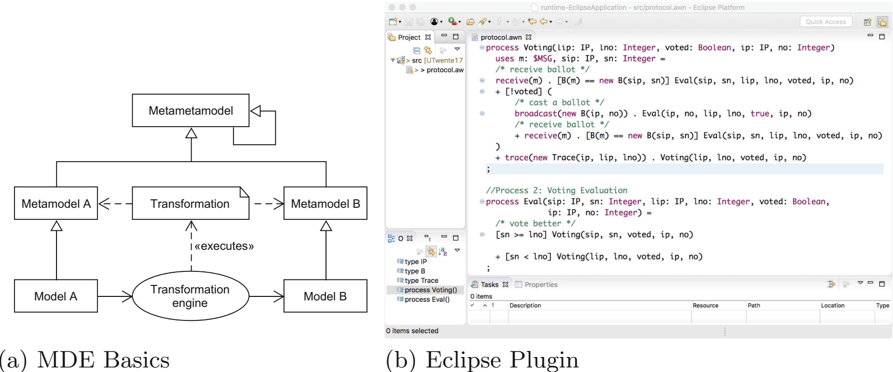
Given Corollary 6.3, the proof is fairly straightforward. As we have established a bisimulation between specifications written in AWN and their translated counterparts in mCRL2 we can now use the mCRL2 toolset to analyse safety properties, because such properties are preserved under bisimulation.

7 Implementation

We have implemented our translation as an Eclipse-plugin, available at http://hoefner-online.de/ifm18/ with a description in [36]. The project, written in Java, is based on the principles of *Model-Driven Engineering* (MDE) [24,33].

MDE efficiently combines domain-specific languages with transformation engines and generators. The MDE approach aims to increase productivity by maximising compatibility between systems, via reuse of standardised models; its basic design principle is sketched in Fig. 2(a). Based on the idea of "everything is a model", the overall goal is to transform a model A into another model B. The syntax of a model is usually a domain-specific language (DSL), or in terms of MDE a metamodel. All metamodels have to conform to the syntax of a metametamodel—the syntax of a metametamodel can be expressed by the metametamodel itself. Using a commonly available metametamodel, such as the

[4] P is an encapsulated network expression when it has the form $[M]$.



(a) MDE Basics (b) Eclipse Plugin

Fig. 2. Implementing AWN to mCRL2

one introduced by the Object Management Group [34], makes metamodels compatible. Abstract transformation rules defined between metamodels are used to transform models.

In our setting Model A is an AWN-specification, and Model B its translated counterpart in mCRL2. The metamodels A and B correspond to the syntax of the two process algebras.

To ensure usability and compatibility, our implementation builds on existing MDE frameworks and techniques. We use the *Eclipse Modeling Framework* (EMF) [35], which includes a metametamodel (Ecore) for describing metamodels. The open-source framework *Xtext* [4] provides infrastructure to create parsers, linkers, and typecheckers. By using Eclipse and Xtext, we are able to provide a user-friendly GUI; see Fig. 2(b). To define our model transformation we use *QVT* (Query/View/Transformation) [27,28]; in particular the imperative model transformation language QVTo.

For development purposes and as a sanity check we implement and translate the leader election protocol, presented in Sects. 2 and 3. Both the input and output are small enough to be manually inspected and analysed.

We use the mCRL2 toolset—in particular the provided model checker—to determine whether the nodes of a 5-node network eventually agree on a leader. In terms of CTL [9], we want to check

$$\mathsf{A}\Diamond\ (\varphi_{\mathtt{lips}}) \qquad \text{and} \qquad \mathsf{A}\Diamond\ \mathsf{A}\Box\ (\varphi_{\mathtt{lips}}),$$

where $\varphi_{\mathtt{lips}}$ is a propositional (state) formula checking the equality of the values assigned to the nodes' variables $\mathtt{lip}$. The former equation states that at some point in time all nodes agree on a common leader; the latter strengthens the statement and requires that the nodes agree on a leader permanently. The mCRL2 toolset only checks formulas written in the modal μ-calculus [32], or (generalised) Hennessy-Milner logic [19]; so we have to translate the above formula. Moreover, because state variables are hidden from direct analysis by mCRL2, we modify the protocol by including an extra, otherwise inert, action **trace**($\mathtt{ip}, \mathtt{lip}$). It reveals the current choice of leader of a particular node, when

added as a 'self-loop' to Process 1. The property can then be specified by requiring that all traces that exclude **trace**-actions eventually can only do exactly one **trace** action for each `ip` with matching values for the `lip` argument. As this is the case, the protocol is correct.

We now analyse a variant of the leader election protocol in which the operator $\geq$ in Process 2 (Line 1) is replaced by $>$, and $<$ by $\leq$. Interestingly, the property under consideration does *not* hold. We leave it to the reader to find the reason.

8 Case Study: The AODV Routing Protocol

To further test our framework, we translate the Ad hoc On-Demand Distance Vector (AODV) routing protocol [30], which two of the authors together with colleagues formalised in AWN before [12,16].

AODV is a reactive protocol, i.e., routes are established on demand, only when needed. It is a widely-used routing protocol designed for Mobile Ad-hoc Networks (MANETs) and Wireless Mesh Networks (WMNs). The protocol is one of the four protocols standardised by the IETF MANET working group, and forms the basis of new WMN protocols, including the Hybrid Wireless Mesh Protocol (HWMP) in the IEEE 802.11s wireless mesh network standard [22].

The AODV routing protocol is specified in the form of an RFC [30], which is the de facto standard. However, it has been shown that the standard contains several ambiguities, contradictions, and cases of underspecification [12].

To overcome these deficiencies, two of the authors, together with other colleagues, obtained the first rigorous formalisation of the AODV routing protocol [12,16], using the process algebra AWN. The specification consists of about 150 lines of AWN-code, split over seven processes, and around 35 functions working on a customised data structure, including routing tables.

The specification, which is available online, is translated into mCRL2, using our framework. It is not the purpose of this paper to perform a proper analysis of this protocol; we merely illustrate the potential of our framework, namely that it can be used to analysis protocols used in modern networks.

Using the translated specification, we analyse a very weak form of the *packet delivery* property [12], which, in generalised Hennessy-Milner logic, is described as

$$[\text{true}^{*}\cdot\mathbf{trace}(\texttt{newpkt}(\texttt{dip},\texttt{data}))]\ [\neg(\mathbf{deliver}(\texttt{dip},\texttt{data})^{*}]\ \langle\text{true}^{*}\rangle \\ \langle\mathbf{deliver}(\mathit{dip},\mathit{data})\rangle\ \text{true}\ .$$

The property states that whenever a new packet intended for `dip` is injected to the system, modelled by the action labelled **trace**(`newpkt`(`dip`, `data`)), then, as long as it has not been delivered yet, it remains possible that this very packet will be delivered in the future. AODV uses a series of control messages to establish a route between source and destination before actually sending the `data`-packet. Similar to the leader election protocol, the AODV specification is modified by inserting a **trace**-action that makes the detection of a `newpkt`-submission to the AODV process visible to mCRL2.

We model a static linear network of three nodes and manually insert two new packets. The mCRL2 toolset checks the packet-delivery property against the given network, and detects a counter example showing that AODV control messages can interfere. Thus, the packet-delivery property does not hold, confirming an analysis done by pen-and-paper [12].

9 Conclusion

In this paper we have developed and implemented a translation from the process algebra AWN into the process algebra mCRL2. The translation allows an automatic analysis of AWN-specifications, using the sophisticated toolset of mCRL2. In contrast to many approaches that transform one formalism into another, we have proven that the translation respects strong bisimilarity (modulo renaming). Therefore we guarantee that all safety properties can be checked on the translated specification and that the (positive/negative) outcome carries over to the AWN-specification. Besides, establishing the relationship in a formal way helped us in finding problems in our translation that we otherwise would have missed. For example, in an early version of the translation function we missed the introduction of a fresh variable y when translating an assignment (Line 7 of Table 4).

We have used our framework, which is available online, to analyse a simple leader election protocol, as well as the packet-delivery property of the AODV routing protocol.

Directions for future work are manifold. (a) Having tools for automated analysis available, we can now analyse further protocols, such as a fragmentation and reassembly protocol running on top of a CAN-bus [15]. (b) T-AWN is an extension of AWN by timing constructs. [7] Since mCRL2 supports time as well, it would be interesting to extend our translation of Sect. 5. Since the timing constructs of T-AWN are fairly complex, this may be a challenging task; in particular if the result on strong bisimulation should be maintained. (c) We want to make use of more highly sophisticated off-the-shelf tools, such as Isabelle/HOL and Uppaal. As our framework follows the methodology of Model-Driven Engineering, implementing translations into other formalisms should be straightforward—proving a bisimulation result is a different story, though. (d) It has been shown that bisimulations do not preserve all liveness properties. [14] We want to come up with a concept that preserves both safety and liveness properties, followed by an adaptation of our translation.

References

1. Behrmann, G., David, A., Larsen, K.G., Pettersson, P., Wang Yi: Developing Uppaal over 15 years. Softw. - Pract. Exp. **41**(2), 133–142 (2011). https://doi.org/10.1002/spe.1006
2. 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). https://doi.org/10.1007/978-3-540-30080-9_7

3. Bergstra, J.A., Klop, J.W.: Algebra of communicating processes. In: de Bakker, Hazewinkel, J.W., Lenstra, J.K. (eds.) Mathematics and Computer Science, CWI Monograph 1, pp. 89–138. North-Holland (1986)
4. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, 2nd edn. Packt Publishing, Birmingham (2016)
5. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Comput. Netw. **14**, 25–59 (1987). https://doi.org/10.1016/0169-7552(87)90085-7
6. Bourke, T., van Glabbeek, R.J., Höfner, P.: Mechanizing a process algebra for network protocols. J. Autom. Reason. **56**(3), 309–341 (2016). https://doi.org/10.1007/s10817-015-9358-9
7. Bres, E., van Glabbeek, R., Höfner, P.: A timed process algebra for wireless networks with an application in routing. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 95–122. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_5
8. Cranen, S., et al.: An overview of the mCRL2 toolset and its recent advances. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 199–213. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_15
9. Emerson, E.A., Clarke, E.M.: Using branching time temporal logic to synthesize synchronization skeletons. Sci. Comput. Program. **2**(3), 241–266 (1982). https://doi.org/10.1016/0167-6423(83)90017-5
10. Fehnker, A., van Glabbeek, R., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: Automated analysis of AODV using UPPAAL. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 173–187. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_13
11. Fehnker, A., van Glabbeek, R., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 295–315. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_15
12. Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A.K., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks used for modelling, verifying and analysing AODV (2013). http://arxiv.org/abs/1312.7645
13. Glabbeek, R.J.: The coarsest precongruences respecting safety and liveness properties. In: Calude, C.S., Sassone, V. (eds.) TCS 2010. IAICT, vol. 323, pp. 32–52. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15240-5_3
14. van Glabbeek, R.J.: Ensuring liveness properties of distributed systems (a research agenda). Position paper (2016). https://arxiv.org/abs/1711.04240
15. van Glabbeek, R.J., Höfner, P.: Split, send, reassemble: a formal specification of a CAN bus protocol stack. In: Hermanns, H., Höfner, P. (eds.) Models for formal analysis of real systems (MARS 2017), EPTCS, vol. 244, pp. 14–52. Open Publishing Association (2017). https://doi.org/10.4204/EPTCS.244.2
16. van Glabbeek, R.J., Höfner, P., Portmann, M., Tan, W.L.: Modelling and verifying the AODV routing protocol. Distrib. Comput. **29**(4), 279–315 (2016). https://doi.org/10.1007/s00446-015-0262-7
17. van Glabbeek, R.J., Höfner, P., van der Wal, D.: Analysing AWN-specifications using mCRL2. Technical report, Data61, CSIRO (2018, to appear)
18. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press, Cambridge (2014)
19. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. J. ACM **32**(1), 137–161 (1985). https://doi.org/10.1145/2455.2460

20. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Upper Saddle River (1985)
21. Hwong, Y.-L., Keiren, J.A., Kusters, V.J.J., Leemans, S.J.J., Willemse, T.A.C.: Formalising and analysing the control software of the Compact Muon Solenoid Experiment at the Large Hadron Collider. Sci. Comput. Program. **78**(12), 2435–2452 (2013). https://doi.org/10.1016/j.scico.2012.11.009
22. IEEE: IEEE Standard for Information Technology—Telecommunications and information exchange between systems—Local and metropolitan area networks-Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications Amendment 10: Mesh Networking (2011). http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6018236
23. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
24. Kent, S.: Model driven engineering. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 286–298. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-47884-1_16
25. Luttik, S.P.: Description and formal specification of the link layer of P1394. In: Lovrek, I. (ed.) 2nd International Workshop on Applied Formal Methods in System Design, pp. 43–56 (1997)
26. Milner, R.: Communication and Concurrency. Prentice Hall, Upper Saddle River (1989)
27. Nolte, S.: QVT - Operational Mappings: Modellierung mit der Query Views Transformation. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-540-92293-3
28. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification (2011). http://www.omg.org/spec/QVT/
29. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981). https://doi.org/10.1007/BFb0017309
30. Perkins, C.E., Belding-Royer, E.M., Das, S.: Ad hoc On-Demand Distance Vector (AODV) routing. RFC 3561 (Experimental), Network Working Group (2003). http://www.ietf.org/rfc/rfc3561.txt
31. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. **60**–**61**, 17–139 (2004). Originally appeared in 1981. https://doi.org/10.1016/j.jlap.2004.05.001
32. Pratt, V.R.: A Decidable mu-Calculus. In: Foundations of Computer Science (FOCS 1981), pp. 421–427. IEEE Computer Society (1981). https://doi.org/10.1109/SFCS.1981.4
33. Schmidt, D.C.: Model-driven engineering. Computer **39**(2), 25–31 (2006). https://doi.org/10.1109/MC.2006.58
34. Soley, R., The OMG Staff Strategy Group: Model Driven Architecture (2000). http://www.omg.org/~soley/mda.html
35. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley, Boston (2009)
36. van der Wal, D.: Modeling AWN networks with an mCRL2 back end. Master's thesis, University of Twente (2018)

Author Index

Ábrahám, Erika 316

Back, Ralph-Johan 151
Basile, Davide 20
Boerman, Jan 30
Bohlender, Dimitri 47

Cavalcanti, Ana 1
Chen, Tzu-Chun 214
Cheng, Zheng 69

Derrick, John 110
Dimjašević, Marko 89
Doherty, Simon 110
Dongol, Brijesh 110
Dougherty, Daniel J. 130

Eriksson, Johannes 151

Falcone, Yliès 255
Fantechi, Alessandro 20
Farrell, Marie 161
Ferrari, Alessio 20
Fisher, Michael 161
Frappier, Marc 377

Galpin, Vashti 172
Gnesi, Stefania 20
Guttman, Joshua D. 130

Höfner, Peter 398
Howar, Falk 89
Huang, Li 236
Huisman, Marieke 30

Jaber, Mohamad 255
Johnsen, Einar Broch 194
Joosten, Sebastiaan 30

Kamburjan, Eduard 214
Kang, Eun-Young 236
Kobeissi, Salwa 255
Körner, Philipp 275
Kowalewski, Stefan 47, 367
Krings, Sebastian 346

Laleau, Régine 377
Lanotte, Ruggero 296
Leofante, Francesco 316
Leuschel, Michael 275, 346, 377
Li, Wei 1
Luckcuck, Matt 161
Luckow, Kasper 89

Mammar, Amel 377
Mazzanti, Franco 20
Meijer, Jeroen 275
Merro, Massimo 296
Miyazawa, Alvaro 1
Mu, Dongrui 236

Nguyen, Huu-Vu 326

Parsa, Masoumeh 151
Piattino, Andrea 20

Rakamarić, Zvonimir 89
Ramsdell, John D. 130
Ribeiro, Pedro 1
Royer, Jean-Claude 69

Sampaio, Augusto 1
Schmidt, Joshua 346
Simon, Hendrik 367
Steffen, Martin 194
Stumpf, Johanna Beate 194

Tacchella, Armando 316
ter Beek, Maurice H. 20
Timmis, Jon 1

Tini, Simone 296
Tisi, Massimo 69
Touili, Tayssir 326
Trentini, Daniele 20
Tueno Fotso, Steve Jeffrey 377
Tveito, Lars 194

Utayim, Adnan 255

van der Wal, Djurre 398
van Glabbeek, Rob 398

Wehrheim, Heike 110