

Formal Methods

LNCS 10811

Aaron Dutle  
César Muñoz  
Anthony Narkawicz (Eds.)

# NASA Formal Methods

**10th International Symposium, NFM 2018  
Newport News, VA, USA, April 17–19, 2018  
Proceedings**



Springer

*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>

Aaron Dutle · César Muñoz  
Anthony Narkawicz (Eds.)

# NASA Formal Methods

10th International Symposium, NFM 2018  
Newport News, VA, USA, April 17–19, 2018  
Proceedings

*Editors*

Aaron Dutle  
NASA Langley Research Center  
Hampton, VA  
USA

Anthony Narkawicz  
NASA Langley Research Center  
Hampton, VA  
USA

César Muñoz  
NASA Langley Research Center  
Hampton, VA  
USA

ISSN 0302-9743                      ISSN 1611-3349 (electronic)  
Lecture Notes in Computer Science  
ISBN 978-3-319-77934-8              ISBN 978-3-319-77935-5 (eBook)  
<https://doi.org/10.1007/978-3-319-77935-5>

Library of Congress Control Number: 2018937364

LNCS Sublibrary: SL2 – Programming and Software Engineering

© Springer International Publishing AG, part of Springer Nature 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.

Printed on acid-free paper

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

# Preface

The NASA Formal Methods (NFM) Symposium is a forum to foster collaboration between theoreticians and practitioners from NASA, academia, and industry, with the goal of identifying challenges and providing solutions to achieving assurance in mission- and safety-critical systems. Examples of such systems include advanced separation assurance algorithms for aircraft, next-generation air transportation, autonomous rendezvous and docking for spacecraft, autonomous on-board software for unmanned aerial systems (UAS), UAS traffic management, autonomous robots, and systems for fault detection, diagnosis, and prognostics. The topics covered by the NASA Formal Methods Symposia include:

- Formal verification, including theorem proving, model checking, and static analysis
- Advances in automated theorem proving including SAT and SMT solving
- Use of formal methods in software and system testing
- Run-time verification
- Techniques and algorithms for scaling formal methods such as abstraction and symbolic methods, compositional techniques, as well as parallel and/or distributed techniques
- Code generation from formally verified models
- Safety cases and system safety
- Formal approaches to fault tolerance
- Theoretical advances and empirical evaluations of formal methods techniques for safety-critical systems, including hybrid and embedded systems
- Formal methods in systems engineering and model-based development
- Formalization of mathematics and physics

This volume contains the papers presented at NFM 2018, the 10th NASA Formal Methods Symposium, held during April 17–19, 2018 in Newport News, VA. NFM 2018 celebrated 30 years of formal methods research at NASA. Previous symposia were held in Moffett Field, CA (2017), Minneapolis, MN (2016), Pasadena, CA (2015), Houston, TX (2014), Moffett Field, CA (2013), Norfolk, VA (2012), Pasadena, CA (2011), Washington, DC (2010), and Moffett Field, CA (2009). The series started as the Langley Formal Methods Workshop, and was held under that name in 1990, 1992, 1995, 1997, 2000, and 2008.

Papers were solicited for NFM 2018 under two categories: regular papers describing fully developed work and complete results, and short papers describing tools, experience reports, or work in progress with preliminary results. The symposium received 92 submissions for review out of which 31 were accepted for publication. These submissions went through a rigorous reviewing process, where each paper was first independently reviewed by at least three reviewers and then subsequently discussed by the Program Committee. In addition to the refereed papers, the symposium featured

two invited presentations, one by Rick Butler of NASA Langley Research Center, USA, and one by Gilles Dowek of Inria, CNRS, and ENS Cachan, France.

The organizers are grateful to the authors for submitting their work to NFM 2018 and to the invited speakers for sharing their insights. NFM 2018 would not have been possible without the collaboration of the outstanding Program Committee and additional reviewers, the support of the Steering Committee, the efforts of the staff at the NASA Langley Research Center, and the general support of the NASA Formal Methods community. The NFM 2018 website can be found at: <https://shemesh.larc.nasa.gov/NFM2018>.

April 2018

Aaron Dutle  
César Muñoz  
Anthony Narkawicz

# Organization

## Program Committee

Erika Ábrahám	RWTH Aachen University, Germany
Mauricio Ayala-Rincon	Universidade de Brasilia, Brazil
Julia Badger	NASA, USA
Dirk Beyer	LMU Munich, Germany
Nikolaj Bjørner	Microsoft, USA
Jasmin Blanchette	Vrije Universiteit Amsterdam, The Netherlands
Sylvie Boldo	Inria, France
Kalou Cabrera Castillos	LAAS-CNRS, France
Misty Davies	NASA, USA
Catherine Dubois	ENSIIE-Samovar, France
Aaron Dutle	NASA, USA
Stefania Gnesi	ISTI-CNR, Italy
Alberto Griggio	Fondazione Bruno Kessler, Italy
George Hagen	NASA, USA
John Harrison	Intel, USA
Klaus Havelund	NASA Jet Propulsion Laboratory, USA
Ashlie Hocking	Dependable Computing, USA
Susmit Jha	SRI International, USA
Rajeev Joshi	NASA Jet Propulsion Laboratory, USA
Laura Kovacs	Vienna University of Technology, Austria
Michael Lowry	NASA, USA
Panagiotis Manolios	Northeastern University, USA
Shaun McWherter	NASA, USA
César Muñoz	NASA, USA
Anthony Narkawicz	NASA, USA
Natasha Neogi	NASA, USA
Lee Pike	Groq, USA
Murali Rangarajan	The Boeing Company, USA
Elvinia Riccobene	University of Milan, Italy
Camilo Rocha	Pontificia Universidad Javeriana Cali, Colombia
Kristin Yvonne Rozier	Iowa State University, USA
Sriram Sankaranarayanan	University of Colorado Boulder, USA
Johann Schumann	SGT, USA
Konrad Slind	Rockwell Collins, USA
Cesare Tinelli	The University of Iowa, USA
Laura Titolo	National Institute of Aerospace, USA



Christoph Torens	German Aerospace Center, Germany
Michael Whalen	University of Minnesota, USA
Virginie Wiels	ONERA, France

## **Additional Reviewers**

Alves, Vander	Katz, Guy
Arcaini, Paolo	Kremer, Gereon
Basile, Davide	Lammich, Peter
Bozzano, Marco	Larraz, Daniel
Braghin, Chiara	Lemberger, Thomas
Brotherston, James	Li, Jianwen
Byun, Taejoon	Lüdtke, Daniel
Chakarov, Aleksandar	Marché, Claude
Champion, Adrien	Marechal, Alexandre
Chen, Xin	Mazzanti, Franco
Chowdhury, Omar	Meel, Kuldeep S.
Cohen, Cyril	Merz, Stephan
Cox, Arlen	Moscato, Mariano
Cruanes, Simon	Nantes-Sobrinho, Daniele
Dangl, Matthias	Nigam, Vivek
Dodds, Mike	Panizo, Laura
Dureja, Rohit	Paskevich, Andrei
Fainekos, Georgios	Pérez, Jorge A.
Fantechi, Alessandro	Ravitch, Tristian
Feliú Gabaldon, Marco	Rioboo, Renaud
Ferrari, Alessio	Roveri, Marco
Fleury, Mathias	Schirmer, Sebastian
Fokkink, Wan	Schopferer, Simon
Friedberger, Karlheinz	Schupp, Stefan
Gallois-Wong, Diane	Stewart, Danielle
Ghassabani, Elaheh	Strub, Pierre-Yves
Goodloe, Alwyn	Tian, Chun
Hoxha, Bardh	Traytel, Dmitriy
Hussein, Soha	Weaver, Sean
Jakobs, Marie-Christine	Wendler, Philipp
Jones, Benjamin	Weps, Benjamin
Katis, Andreas	

# Contents

Incremental Construction of Realizable Choreographies . . . . .	1
<i>Sarah Benyagoub, Meriem Ouederni, Yamine Aït-Ameur, and Atif Mashkoor</i>	
Formal Assurance for Cooperative Intelligent Autonomous Agents . . . . .	20
<i>Siddhartha Bhattacharyya, Thomas C. Eskridge, Natasha A. Neogi, Marco Carvalho, and Milton Stafford</i>	
Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C . . . . .	37
<i>Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue</i>	
An Executable Formal Framework for Safety-Critical Human Multitasking. . .	54
<i>Giovanna Broccia, Paolo Milazzo, and Peter Csaba Ölveczky</i>	
Simpler Specifications and Easier Proofs of Distributed Algorithms Using History Variables. . . . .	70
<i>Saksham Chand and Yanhong A. Liu</i>	
Don't Miss the End: Preventing Unsafe End-of-File Comparisons . . . . .	87
<i>Charles Zhuo Chen and Werner Dietl</i>	
An Efficient Rewriting Framework for Trace Coverage of Symmetric Systems . . . . .	95
<i>Flavio M. De Paula, Arvind Haran, and Brad Bingham</i>	
Verification of Fault-Tolerant Protocols with Sally . . . . .	113
<i>Bruno Dutertre, Dejan Jovanović, and Jorge A. Navas</i>	
Output Range Analysis for Deep Feedforward Neural Networks . . . . .	121
<i>Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari</i>	
Formal Dynamic Fault Trees Analysis Using an Integration of Theorem Proving and Model Checking . . . . .	139
<i>Yassmeen Elderhalli, Osman Hasan, Waqar Ahmad, and Sofiène Tahar</i>	
Twenty Percent and a Few Days – Optimising a Bitcoin Majority Attack. . .	157
<i>Ansgar Fehnker and Kaylash Chaudhary</i>	
An Even Better Approach – Improving the B.A.T.M.A.N. Protocol Through Formal Modelling and Analysis . . . . .	164
<i>Ansgar Fehnker, Kaylash Chaudhary, and Vinay Mehta</i>	

Towards a Formal Safety Framework for Trajectories . . . . .	179
<i>Marco A. Feliù and Mariano M. Moscato</i>	
Static Value Analysis of Python Programs by Abstract Interpretation . . . . .	185
<i>Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné</i>	
Model-Based Testing for General Stochastic Time. . . . .	203
<i>Marcus Gerhold, Arnd Hartmanns, and Mariëlle Stoelinga</i>	
Strategy Synthesis for Autonomous Agents Using PRISM . . . . .	220
<i>Ruben Giaquinta, Ruth Hoffmann, Murray Ireland, Alice Miller, and Gethin Norman</i>	
The Use of Automated Theory Formation in Support of Hazard Analysis. . . . .	237
<i>Andrew Ireland, Maria Teresa Llano, and Simon Colton</i>	
Distributed Model Checking Using PROB. . . . .	244
<i>Philipp Körner and Jens Bendisposto</i>	
Optimal Storage of Combinatorial State Spaces. . . . .	261
<i>Alfons Laarman</i>	
Stubborn Transaction Reduction . . . . .	280
<i>Alfons Laarman</i>	
Certified Foata Normalization for Generalized Traces . . . . .	299
<i>Hendrik Maarand and Tarmo Uustalu</i>	
On the Timed Analysis of Big-Data Applications . . . . .	315
<i>Francesco Marconi, Giovanni Quattrocchi, Luciano Baresi, Marcello M. Bersani, and Matteo Rossi</i>	
Tuning Permissiveness of Active Safety Monitors for Autonomous Systems . . . . .	333
<i>Lola Masson, Jérémie Guiochet, Hélène Waeselynck, Kalou Cabrera, Sofia Cassel, and Martin Törngren</i>	
Sound Black-Box Checking in the LearnLib. . . . .	349
<i>Jeroen Meijer and Jaco van de Pol</i>	
Model-Checking Task Parallel Programs for Data-Race . . . . .	367
<i>Radha Nakade, Eric Mercer, Peter Aldous, and Jay McCarthy</i>	
Consistency of Property Specification Patterns with Boolean and Constrained Numerical Signals . . . . .	383
<i>Massimo Narizzano, Luca Pulina, Armando Tacchella, and Simone Vuotto</i>	

Automatic Generation of DO-178 Test Procedures . . . . .	399
<i>César Ochoa Escudero, Rémi Delmas, Thomas Bochot, Matthieu David, and Virginie Wiels</i>	
Using Test Ranges to Improve Symbolic Execution. . . . .	416
<i>Rui Qiu, Sarfraz Khurshid, Corina S. Păsăreanu, Junye Wen, and Guowei Yang</i>	
Symbolic Execution and Reachability Analysis Using Rewriting Modulo SMT for Spatial Concurrent Constraint Systems with Extrusion . . . . .	435
<i>Miguel Romero and Camilo Rocha</i>	
Experience Report: Application of Falsification Methods on the UxAS System. . . . .	452
<i>Cumhur Erkan Tuncali, Bardh Hoxha, Guohui Ding, Georgios Fainekos, and Sriram Sankaranarayanan</i>	
MoDeS3: Model-Based Demonstrator for Smart and Safe Cyber-Physical Systems . . . . .	460
<i>András Vörös, Márton Búr, István Ráth, Ákos Horváth, Zoltán Micskei, László Balogh, Bálint Hegyi, Benedek Horváth, Zsolt Mázló, and Dániel Varró</i>	
<b>Author Index</b> . . . . .	469



# Incremental Construction of Realizable Choreographies

Sarah Benyagoub<sup>1,2</sup>, Meriem Ouederni<sup>2</sup>, Yamine Aït-Ameur<sup>2(✉)</sup>,  
and Atif Mashkoor<sup>3</sup>

<sup>1</sup> University of Mostaganem, Mostaganem, Algeria  
benyagoub.sarah@univ-mosta.dz

<sup>2</sup> INP-ENSEEIH/IRIT, Université de Toulouse, Toulouse, France  
{ouederni,yamine}@enseeiht.fr

<sup>3</sup> SCCH GmbH and Johannes Kepler University, Linz, Austria  
atif.mashkoor@scch.at, atif.mashkoor@jku.at

**Abstract.** This paper proposes a correct-by-construction method to build realizable choreographies described using conversation protocols (CPs). We define a new language consisting of an operators set for incremental construction of CPs. We suggest an asynchronous model described with the Event-B method and its refinement strategy, ensuring the scalability of our approach.

**Keywords:** Realisability · Conversation protocols  
Correct-by-construction method proof and refinement · Event-B

## 1 Introduction

Distributed systems are pervasive in areas like embedded systems, Cyber Physical systems, medical devices and Web applications. In a top-down design of such systems, the interaction among peers is usually defined using a global specification called conversation protocols (CP), aka choreography in SOC [9]. These CPs specify interactions among peers as the allowed sequences of sent messages.

A main concern, already addressed by research community, is the verification of CP realizability i.e., verification whether *there exists a set of peers whose composition generates the same sequences of sending messages as specified by the CP*. Considering asynchronous communication, this realizability problem is undecidable in general [8] due to possible ever-increasing queuing mechanism and unbounded buffers. The work of [5] proposed a necessary and sufficient condition for verifying whether a CP can be implemented by a set of peers communicating asynchronously using FIFO buffers with no buffer sizes restrictions. This

---

The research reported in this paper has been partly supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

work solves the realizability issue for a subclass of asynchronously communicating peers (synchronizable systems) *i.e.*, systems composed of interacting peers behaving equivalently either with synchronous or asynchronous communication.

A CP is *realizable* if there exists a set of peers implementing this CP, *i.e.*, the peers send messages to each other in the same order as the CP does, and their composition is synchronizable. In [5], checking CP realizability applies three steps: (i) *peer projection* from CP; (ii) checking *synchronizability*; and (iii) checking *equivalence* between CP and its distributed system obtained after projection.

The work given in [5] relies on model checking for systems with reasonable sizes (*i.e.*, number of states, transitions and communicating peers). This verification procedure is global and a posteriori. It considers the whole CP and its projection, and does not handle compositional verification.

This paper proposes a *compositional* and *incremental* formal verification procedure that scales to systems of arbitrary sizes. It promotes a top-down design of realizable CPs following a correct-by-construction method which decreases the complexity of the verification task and supports real-world complex systems. We define a compositional language using an algebra of operators (sequence, branching, and loop). From an initial basic CP, we inductively (incrementally) build a realizable CP by composing other realizable ones, using these composition operators while preserving realizability [5] w.r.t identified conditions. The informal definition of these operators were originally introduced in [6, 7] the feasibility of the approach on toy case studies is shown. [6, 7] did not give the formal proof of correctness of realizability preservation of the defined operators. Consequently, in this paper, we provide a correctness support for the results sketched in [6, 7]. An inductive proof, based on realizability invariant preservation, is set up with Event-B [2] on Rodin [19] platform. Refinement is used to decompose this invariant in order to ease the proof and development processes. The generic model we define is scalable and its parameters have arbitrary values (*i.e.*, numbers of peers, buffer sizes, number of states and transitions can take any value in their corresponding sets of possible values). Furthermore, this model can be instantiated to describe any CP by incremental application of the composition operators we defined.

In the remainder, Sect. 2 introduces the formal definitions and the background our proposal relies on. Section 3 presents the set of composition operators together with the set of identified sufficient conditions that ensure realizability of the built CPs. The formal Event-B development based on the refinement strategy we have set up is shown in Sect. 4. Finally, Sect. 5 overviews related work Sect. 6 concludes this work.

## 2 Background and Notations

### 2.1 Model

We use labeled transition systems (LTSs) for modeling CP and peers included in that specification. This model defines messages order being sent in CP.

**Definition 1 (Peer).** A peer is an LTS  $\mathcal{P} = (S, s^0, \Sigma, T)$  where  $S$  is a finite set of states,  $s^0 \in S$  is the initial state,  $\Sigma = \Sigma^! \cup \Sigma^? \cup \{\tau\}$  is a finite alphabet partitioned into a set of send messages, receive messages, and the internal action, and  $T \subseteq S \times \Sigma \times S$  is a transition relation.

We write  $m!$  for a send message  $m \in \Sigma^!$  and  $m?$  for a receive message  $m \in \Sigma^?$ . We use the symbol  $\tau$  for representing internal activities. A transition is represented as  $s \xrightarrow{l} s'$  where  $l \in \Sigma$ . Notice that we refer to a state  $s^f \in S$  as final if there is no outgoing transition at that state.

**Definition 2 (CP).** A conversation protocol CP for a set of peers  $\{\mathcal{P}_1, \dots, \mathcal{P}_n\}$  is a LTS  $CP = (S_{CP}, s_{CP}^0, L_{CP}, T_{CP})$  where  $S_{CP}$  is a finite set of states and  $s_{CP}^0 \in S_{CP}$  is the initial state;  $L_{CP}$  is a set of labels where a label  $l \in L_{CP}$  is denoted  $m^{\mathcal{P}_i, \mathcal{P}_j}$  such that  $\mathcal{P}_i$  and  $\mathcal{P}_j$  are the sending and receiving peers, respectively,  $\mathcal{P}_i \neq \mathcal{P}_j$ , and  $m$  is a message on which those peers interact; finally,  $T_{CP} \subseteq S_{CP} \times L_{CP} \times S_{CP}$  is the transition relation. We require that each message has a unique sender and receiver:  $\forall \{m^{\mathcal{P}_i, \mathcal{P}_j}, m'^{\mathcal{P}'_i, \mathcal{P}'_j}\} \subseteq L_{CP} : m = m' \implies \mathcal{P}_i = \mathcal{P}'_i \wedge \mathcal{P}_j = \mathcal{P}'_j$ .

In the remainder of this paper, we denote a transition  $t \in T_{CP}$  as  $s \xrightarrow{m^{\mathcal{P}_i, \mathcal{P}_j}} s'$  where  $s$  and  $s'$  are source and target states and  $m^{\mathcal{P}_i, \mathcal{P}_j}$  is the transition label. We refer to a basic CP  $= \langle S_{CP}, s_{CP}^0, L_{CP}, T_{CP} \rangle$  as  $CP_b$  if and only if  $T_{CP} = \{S_{CP} \xrightarrow{m^{\mathcal{P}_i, \mathcal{P}_j}} s'_{CP}\}$ . We refer to the set of final states as  $S^f$  where the system can terminate its execution. It is worth noticing that the peers' LTSs are computed by projection from CP as follows:

**Definition 3 (Projection).** Let the projection function  $\downarrow CP$  which returns the set of peers LTSs  $\mathcal{P}_i = \langle S_i, s_i^0, \Sigma_i, T_i \rangle$  obtained by replacing in  $CP = \langle S_{CP}, s_{CP}^0, L_{CP}, T_{CP} \rangle$  each label  $(\mathcal{P}_j, m, \mathcal{P}_k) \in L_{CP}$  with  $m!$  if  $j = i$  with  $m?$  if  $k = i$  and with  $\tau$  (internal action) otherwise; and finally removing the  $\tau$ -transitions by applying standard minimization algorithms [14].

**Definition 4 (Synchronous System).** The synchronous system denoted as  $Sys_{sync}(\mathcal{P}_1, \dots, \mathcal{P}_n) = (S_s, s_s^0, L_s, T_s)$  corresponds to the product of peer LTSs composed under synchronous communication semantics.

In this context, a communication between two peers occurs if both agree on a synchronization label, *i.e.*, if one peer is in a state in which a message can be sent, then the other peer must be in a state in which that message can be received. A peer can evolve independently from others through internal actions.

**Definition 5 (Asynchronous System).** In the asynchronous system denoted as  $Sys_{async}(\mathcal{P}_1, \dots, \mathcal{P}_n) = (S_a, s_a^0, L_a, T_a)$ , peers communication holds through FIFO buffers. Each peer  $\mathcal{P}_i$  is equipped with an unbounded message buffer  $Q_i$ .

Where a peer can either send a message  $m \in \Sigma^!$  to the tail of the receiver buffer  $Q_j$  at any state where this sent message is available, read a message  $m \in \Sigma^?$  from its buffer  $Q_i$  if the message is available at the buffer head, or evolve independently through an internal action. Reading from the buffer is non observable, and it is presented by internal action in the asynchronous system.

## 2.2 Realizability

The definition of realizability we use in this paper is borrowed from [5]. A CP is realizable if there exists a set of peers where their composition generates the same sequences of sending messages as specified in CP. In [5] a defined sufficient and necessary condition characterizes the set  $R \subseteq CP$  of realizable CPs. A deterministic  $cp \in R$  is realizable iff the system obtained from the composition of the projected peers of  $cp$  is *synchronizable*, *well-formed*, and *equivalent* to the initial CP. A proof of correctness of global system realizability using Event-B is available in [13].

**Definition 6 (Deterministic Choice).** *Let  $DC$  be the set of deterministic CPs, thus  $\forall CP \in DC : \forall s_{CP} \in S_{CP}, \nexists \{s_{CP} \xrightarrow{m^{P_i, P_j}} s'_{CP}, s_{CP} \xrightarrow{m^{P_i, P_j}} s''_{CP}\} \subseteq T_{CP}$  where  $s'_{CP} \neq s''_{CP}$ .*

**Definition 7 (Equivalence).** *CP is equivalent to  $Sys_{sync}(\downarrow CP)$ , denoted  $CP \equiv Sys_{sync}(\downarrow CP)$ , if they have equal message sequences, i.e., trace equivalence [16].*

A system is synchronizable when its behavior remains the same under both synchronous and asynchronous communication semantics.

**Definition 8 (Synchronizability).** *Given a set of peers  $\{\mathcal{P}_1, \dots, \mathcal{P}_n\}$ , the synchronous system  $Sys_{sync}(\mathcal{P}_1, \dots, \mathcal{P}_n) = (S_s, s_s^0, L_s, T_s)$ , and the asynchronous system  $Sys_{async}(\mathcal{P}_1, \dots, \mathcal{P}_n) = (S_a, s_a^0, L_a, T_a)$ , two states  $r \in S_s$  and  $s \in S_a$  are synchronizable if there exists a relation  $Sync\_st$  between states such that  $Sync\_st(r, s)$  and:*

- for each  $r \xrightarrow{m} r' \in T_s$ , there exists  $s \xrightarrow{m!} s' \in T_a$ , such that  $Sync\_st(r', s')$ ;
- for each  $s \xrightarrow{m!} s' \in T_a$ , there exists  $r \xrightarrow{m} r' \in T_s$ , such that  $Sync\_st(r', s')$ ;
- for each  $s \xrightarrow{m?} s' \in T_a$ ,  $Sync\_st(r, s')$ .

*Synchronizability is the set of synchronizable systems such that  $Sys_{async}(\mathcal{P}_1, \dots, \mathcal{P}_n) \in Synchronizability \Leftrightarrow Sync\_st(s_s^0, s_a^0)$ .*

*Well-formedness* states that whenever the size of a receive queue,  $Q_i$ , of the  $i^{th}$  peer is greater than 0 (i.e.,  $Q_i$  is non-empty), the asynchronous system can eventually move to a state where  $Q_i$  is empty.

**Definition 9 (Well-formedness).** *Let  $WF$  be the set of well formed system. An asynchronous system  $Sys_{async} = (S_a, s_a^0, \Sigma_a, T_a)$  defined over a set of peers  $\{\mathcal{P}_1, \dots, \mathcal{P}_n\}$  is well-formed, i.e.,  $Sys_{async} \in WF$ , if and only if  $\forall s_a = (s_1, Q_1, \dots, s_n, Q_n) \in S_a$ , where  $s_a$  is reachable from  $s_a^0 = (s_1^0, \epsilon, \dots, s_n^0, \epsilon)$ , the following holds: if there exists  $Q_i$  such that  $|Q_i| > 0$ , then there exists  $s_a \Rightarrow^* s'_a \subseteq T_a$  where  $s'_a = (s'_1, Q'_1, \dots, s'_n, Q'_n) \in S_a$  and  $\forall Q'_i, |Q'_i| = 0$ .*

Note that  $\Rightarrow^*$  means that there exists one or more transitions in the asynchronous system (Definition 5) leading into the state  $s'_a$ .

**Definition 10 (Realizability).**  $\forall CP \in DC : CP \in R \iff (CP \equiv Sys_{sync}(\downarrow CP)) \wedge (Sys_{async}(\downarrow CP) \in Synchronizability) \wedge (Sys_{async}(\downarrow CP) \in WF)$ .



### 3 CCP Language for Realisable CPs

In this section, we define our composition operators and identify the conditions sufficient to build  $CP$  realizable  $CP$ s.

#### 3.1 Composition Operators

We present the proposed composition operators  $\otimes_{(\gg, s_{CP}^f)}$  (sequence),  $\otimes_{(+, s_{CP}^f)}$  (branching), and  $\otimes_{(\cup, s_{CP}^f)}$  (iteration) where  $s_{CP}^f \in S_{CP}^f$ . Each expression of the form  $\otimes_{(op, s_{CP}^f)}(CP, CP_b)$  assumes that the initial state of  $CP_b$  is fused with the final state  $s_{CP}^f$ . In the other word,  $CP_b$  is appended to  $CP$  at state  $s_{CP}^f$ .

**Definition 11.** *Sequential Composition  $\otimes_{(\gg, s_{CP}^f)}$ . Given a  $CP$ , a state  $s_{CP} \in S_{CP}^f$ , and a  $CP_b$  where  $T_{CP_b} = \{s_{CP_b} \xrightarrow{l_{CP_b}} s'_{CP_b}\}$ , the sequential composition  $CP_{\gg} = \otimes_{(\gg, s_{CP}^f)}(CP, CP_b)$  means that  $CP_b$  must be executed after  $CP$  starting from  $s_{CP}$ , and:*

$$\begin{aligned} - S_{CP_{\gg}} &= S_{CP} \cup \{s'_{CP_b} \mid s_{CP_b} \xrightarrow{l_{CP_b}} s'_{CP_b} \in T_{CP_b}\} \\ - L_{CP_{\gg}} &= L_{CP} \cup \{l_{CP_b}\} \\ - T_{CP_{\gg}} &= T_{CP} \cup \{s_{CP} \xrightarrow{l_{CP_b}} s'_{CP_b}\} \\ - S_{CP_{\gg}}^f &= (S_{CP}^f \setminus \{s_{CP}\}) \cup \{s'_{CP_b}\} \end{aligned}$$

**Definition 12.** *Choice Composition  $\otimes_{(+, s_{CP}^f)}$ . Given a  $CP$ , a state  $s_{CP} \in S_{CP}^f$ , a set  $\{CP_{bi} \mid i = [1..n], n \in \mathbb{N}\}$  such that  $\forall T_{CP_{bi}}, T_{CP_{bi}} = \{s_{CP_{bi}} \xrightarrow{l_{CP_{bi}}} s'_{CP_{bi}}\}$ , the branching composition  $CP_+ = \otimes_{(+, s_{CP}^f)}(CP, \{CP_{bi}\})$  means that  $CP$  must be executed before  $\{CP_{bi}\}$  and there is a choice between all  $\{CP_{bi}\}$  at  $s_{CP}$ , and:*

$$\begin{aligned} - S_{CP_+} &= S_{CP} \cup \{s'_{CP_{b1}}, \dots, s'_{CP_{bn}} \mid s_{CP_{bi}} \xrightarrow{l_{CP_{bi}}} s'_{CP_{bi}} \in T_{CP_{bi}}\} \\ - L_{CP_+} &= L_{CP} \cup \{l_{CP_{b1}}, \dots, l_{CP_{bn}}\} \\ - T_{CP_+} &= T_{CP} \cup \{s_{CP} \xrightarrow{l_{CP_{b1}}} s'_{CP_{b1}}, \dots, s_{CP} \xrightarrow{l_{CP_{bn}}} s'_{CP_{bn}}\} \\ - S_{CP_+}^f &= (S_{CP}^f \setminus \{s_{CP}\}) \cup \{s'_{CP_{b1}}, \dots, s'_{CP_{bn}}\} \end{aligned}$$

**Definition 13.** *Loop Composition  $\otimes_{(\cup, s_{CP}^f)}$ . Given  $CP$ , a state  $s_{CP} \in S_{CP}^f$ , and a set  $CP_b$ , such that  $T_{CP_b} = \{s_{CP_b} \xrightarrow{l_{CP_b}} s_{CP_b}\}$ , the loop composition  $CP_{\cup} = \otimes_{(\cup, s_{CP}^f)}(CP, CP_b)$  means that  $CP$  must be executed before  $CP_b$  and every  $CP_b$  can be repeated 0 or more times, and:*

$$\begin{aligned} - S_{CP_{\cup}} &= S_{CP} \\ - L_{CP_{\cup}} &= L_{CP} \cup \{l_{CP_b}\} \\ - T_{CP_{\cup}} &= T_{CP} \cup \{s_{CP} \xrightarrow{l_{CP_{b1}}} s_{CP_b}\} \\ - S_{CP_{\cup}}^f &= S_{CP}^f \end{aligned}$$

### 3.2 Realizable-by-Construction CP

As mentioned in the introduction, our intention is to avoid a posteriori global verification of realisability. We set up an incremental verification of realisability using a correct by construction approach. Building CPs using the aforementioned operators does not guarantee its realisability. Indeed, the definitions of the previous operators rely on syntactic conditions mainly by gluing final and initial states of the composed CPs.

**Sufficient Conditions.** We identified a set of sufficient conditions (*i.e.*, Conditions 1, 2, and 3 which entail realisability when the CPs are built using the operators we have previously defined. These conditions are based on the semantics of the messages ordering and exchange.

**Condition 1 (Deterministic Choice (DC)).** *See Definition 6.*

**Condition 2 (Parallel-Choice Freeness (PCF)).** *Let PCF be the set of CPs free of parallel choice. Then,  $CP \in PCF$  iff  $\forall s_{CP} \in S_{CP}, \nexists \{s_{CP} \xrightarrow{m^{P_i, P_j}} s'_{CP}, s_{CP} \xrightarrow{m^{P_k, P_q}} s''_{CP}\} \subseteq T_{CP}$  such that  $P_i \neq P_k$  and  $s'_{CP} \neq s''_{CP}$ .*

**Condition 3 (Independent Sequences Freeness (IseqF)).** *Let IseqF be the set of CPs free of independent sequences. Then,  $CP \in IseqF$  iff  $\forall s_{CP} \in S_{CP}, \nexists \{s_{CP} \xrightarrow{m^{P_i, P_j}} s'_{CP}, s'_{CP} \xrightarrow{m^{P_k, P_q}} s''_{CP}\} \subseteq T_{CP}$  such that  $P_i \neq P_k$  and  $P_j \neq P_k$ .*

All these conditions are structural conditions defined at the CP level. They do not involve conditions on the synchronous nor on the asynchronous projections.

**Realizable-by-Construction CP Theorems.** Table 1 gives the theorems that ensure the realisability of a CP built incrementally using our composition operators. Each theorem relies on the previously introduced sufficient conditions.

**Proof Sketch.** To prove the theorems of Table 1 we rely on a generic proof pattern consisting in decomposing the realisability condition of Definition 10. According to this definition, we need to prove equivalence (Definition 7), synchronizability (Definition 8) and well formedness (WF in Definition 9).

**Table 1.** Theorems for realizable by construction CPs

<i>Theorem 1</i>	$CP_b \in R$
<i>Theorem 2</i>	$CP \in R \wedge CP_b \in R \wedge CP_{\gg} = \otimes_{(\gg, s_{CP}^f)}(CP, CP_b) \in ISeqF \Rightarrow CP_{\gg} \in R$
<i>Theorem 3</i>	$CP \in R \wedge \{CP_{bi}\} \subseteq R \wedge CP_+ = \otimes_{(+, s_{CP}^f)}(CP, \{CP_{bi}\}) \wedge CP_+ \in DC \wedge CP_+ \in ISeqF \wedge CP_+ \in PCF \Rightarrow CP_+ \in R$
<i>Theorem 4</i>	$CP \in R \wedge CP_b \in R \wedge CP_{\circ} = \otimes_{(\circ, s_{CP}^f)}(CP, CP_b) \in ISeqF \Rightarrow CP_{\circ} \in R$

The proof is a structural induction on the defined operators. Let  $CP_b \in R$  and  $CP \in R$  be a basic realizable CP and a realizable CP respectively. We need to prove that  $CP_{op} \in R$  holds for each composition operator  $op \in \{\gg, +, \cup\}$  when the defined sufficient condition  $op_{cond}$  corresponding to conditions 1, 2 and 3 defined above and associated to each  $op$  holds.

When considering the equivalence, synchronisability and well formedness, this proof uses the projection  $\downarrow CP_{op}$  of  $CP_{op}$ . It can be formalised using the following proof pattern.

$$CP \in R \wedge CP_b \in R \wedge \mathbf{Op}_{cond} \implies \begin{cases} CP_{op} \equiv Sys_{sync}(\downarrow CP_{op}) \\ \wedge \\ Sys_{async}(\downarrow CP_{op}) \in Synchronizability \\ \wedge \\ Sys_{async}(\downarrow CP_{op}) \in WF \end{cases} \quad (1)$$

**Theorem 1.** *Any  $CP_b$  is realizable.*

**Proof 1.**  $CP_b$  is made of a single transition of the form  $s \xrightarrow{m^{P_i, P_j}} s'$ . Therefore, the projection will produce two peers  $\mathcal{P}_i$  and  $\mathcal{P}_j$  with a single transition where  $\mathcal{P}_i$  sends the message  $m$  to the receiving peer  $\mathcal{P}_j$ . This projection is realizable.

**Theorem 2.** *Given an  $CP = \langle S_{CP}, s_{CP}^0, L_{CP}, T_{CP} \rangle$  and a  $CP_b$  such that  $CP \in R$  and  $CP_b \in R$ ,  $s_{CP} \in S_{CP}^f$ , then  $CP_{\gg} = \otimes_{(\gg, s_{CP})}(CP, CP_b) \in R$ .*

**Proof 2.** The proof is inductive. It follows the previous proof pattern. When this pattern is instantiated for the sequence operator, we obtain.

$$CP \in R \wedge CP_b \in R \wedge CP_{\gg} \in \mathbf{ISeqF} \implies \begin{cases} CP_{\gg} \equiv Sys_{sync}(\downarrow CP_{\gg}) & (2.a) \\ \wedge \\ Sys_{async}(\downarrow CP_{\gg}) \in Synchronizability & (2.b) \\ \wedge \\ Sys_{async}(\downarrow CP_{\gg}) \in WF & (2.c) \end{cases} \quad (2)$$

**Basic case.** Let  $CP = \emptyset$  and a  $CP_b$  then  $CP_{\gg} = \otimes_{(\gg, s_{CP}^0)}(\emptyset, CP_b) \in R$ . So  $CP_{\gg} = CP_b$ .  $CP_{\gg} \in R$  holds by Theorem 1 of Table 1.

**Inductive Case.** Let  $CP = \langle S_{CP}, s_{CP}^0, L_{CP}, T_{CP} \rangle$  and a  $CP_b$  such that  $CP \in R$  and  $CP_b \in R$ . Let  $s_{CP} \in S_{CP}^f$  be the gluing state (i.e. both the final state of  $CP$  and the initial state of  $CP_b$ ). Let  $s_i^q$  denote the  $i^{th}$  state in the LTS associated to peer  $P_q$ .

According to the proof schema of Eq. 2, we need to prove the Properties 2.a, 2.b and 2.c

**2.a Equivalence property.** By recurrence hypotheses we write  $CP \equiv Sys_{sync}(\downarrow CP)$ ,  $CP_b \equiv Sys_{sync}(\downarrow CP_b)$ . Let us assume that the sufficient condition for sequence holds i.e.  $CP_{\gg} \in ISeqF$ . We need to prove now that  $CP_{\gg} \equiv Sys_{sync}(\downarrow CP_{\gg})$  (Eq. (1.a)).

Let us consider

- any trace  $T_{CP} = \{s_0 \xrightarrow{m^{P_i \rightarrow P_j}} s_1, \dots, s_n \xrightarrow{m^{P_k \rightarrow P_q}} s_{n+1}\}$  in the realizable  $CP$
- and the trace  $T_{CP_b} = \{s_{b0} \xrightarrow{m''^{P_t \rightarrow P_z}} s_{b1}\}$  in the realizable  $CP_b$

Since the *ISeqF* condition holds, two cases are distinguished.

1. **Either**  $P_k = P_t$ , then the following suffixes of the traces occur for peers  $P_k = P_t$ ,  $P_q$  and  $P_z$ 
  - $\{\dots, s_n^k \xrightarrow{m'!} s_{n+1}^k, s_{n+1}^k \xrightarrow{m''!} s_{n+2}^k\} \subseteq T_k$
  - $\{\dots, s_n^q \xrightarrow{m'??} s_{n+1}^q\} \subseteq T_q$
  - $\{\dots, s_n^z \xrightarrow{m''??} s_{n+1}^z\} \subseteq T_z$ .
2. **or**  $P_q = P_t$ , then the following traces occurs for peers  $P_q = P_t$ ,  $P_k$  and  $P_z$ 
  - $\{\dots, s_n^k \xrightarrow{m'!} s_{n+1}^k\} \subseteq T_k$
  - $\{\dots, s_n^q \xrightarrow{m'??} s_{n+1}^q, s_{n+1}^q \xrightarrow{m''!} s_{n+2}^q\} \subseteq T_q$
  - $\{\dots, s_n^z \xrightarrow{m''??} s_{n+1}^z\} \subseteq T_z$

Thanks to the *ISeqF* property, the sending-receiving transition (synchronous transition) of  $CP_b$  requires that either the sending peer or the receiving peer of the  $CP_b$  are used by the previous transition or the realizable  $CP$ . Moreover, it is always performed once the sending-receiving transitions of the synchronous projection of  $CP$  are completed. The sending-receiving transition of  $CP_b$  becomes the last transition of  $Sys_{sync}(\downarrow CP_{\gg})$ .

**2.b Synchronisability condition.** By the recurrence hypotheses, we write  $Sys_{async}(\downarrow CP) \in Synchronizability$ ,  $Sys_{async}(\downarrow CP_b) \in Synchronizability$ . Synchronisability is deduced from equivalence and from the *ISeqF* condition. The last transition of the traces of  $\downarrow CP_{\gg}$  corresponds to  $Sys_{sync}(\downarrow CP_b) = \{s_{b0} \xrightarrow{m''} s_{b1}\}$  and  $Sys_{async}(\downarrow CP_b) = \{s_{b0} \xrightarrow{m'??} s_b, s_b \xrightarrow{m''??} s_{b1}\}$  where  $S_b$  is an intermediate state in the asynchronous projection. In this intermediate state, in which the queues related to the peers contain the message  $m''$ .

**2.c Well-formedness condition.** Again, as recurrence hypotheses, we write  $Sys_{async}(\downarrow CP) \in WF$ ,  $Sys_{async}(\downarrow CP_b) \in WF$ . This means that by hypotheses, the queues are empty in the final state of  $Sys_{async}(\downarrow CP)$  since it is realizable (thus well formed). We have to show that the queue is still empty after running message exchanges of  $CP_b$ .

When adding a sequence  $\otimes_{(\gg, s_{CP}^f)}(CP, CP_b) \in ISeqF$ , the sending transition of  $m''$  gives  $Q_i = \emptyset, Q_j = \emptyset, Q_k = \emptyset, Q_q = \emptyset, Q_t = \emptyset, Q_z = \{m''\}$ . It and the consumption of the  $m''$  empties the queue  $Q_z$  such that  $Q_i = \emptyset, Q_j = \emptyset, Q_k = \emptyset, Q_q = \emptyset, Q_t = \emptyset, Q_z = \emptyset$ .

At this level we can conclude that the defined sequence composition operator preserves realizability.

The proofs for the choice and loop operators follow the same inductive schema. We do not present these proofs due to space limitations. A sketch of these proofs is given in [6].

## 4 CCP Model: Refinement-Based Realizability

The proofs reported in the previous section are handmade. In order to give full confidence in our results on correct-by-construction realizability, we designed a whole formal development of this proof using refinement. The Event-B method has been set up as follows.

### 4.1 The Refinement Strategy

The refinement operator offered by the Event-B method proved efficient to handle the complex proofs associated to each operators. This operator allowed us to handle the realizability property incrementally by introducing first equivalence, then synchronizability and finally well formedness in specific refinements. Therefore, the following refinement strategy has been set up:

**Root Model.** The root model defines the conversation protocols. It introduces basic CP. Each composition operator is defined as an event which incrementally builds the final CP obtained by introducing a final state. All the built CP satisfy an invariant requiring DC (deterministic choice, Condition 1). This model also declares a prophecy variable [1] as a state variable. This variable defines an arbitrary numbers of exchanged messages and is used to define a variant in order to further prove well formedness.

**First Refinement: The Synchronous Model.** The second model is obtained by refining each event (composition operator) to define the synchronous projection. A gluing invariant linking the CP to the synchronous projection is introduced. The equivalence property is proved at this level. It is defined as an invariant preserved by all the events encoding a composition operator. This projection represents the synchronous system, it preserves the message exchanges order between peers and hides the asynchronous exchanges.

**Second Refinement: The Asynchronous Model.** The last model introduces the asynchronous projection. Each event (composition operator) is refined to handle the asynchronous communication. Synchronous and asynchronous projections are linked by another gluing invariant. Sending and receiving actions together with queue handling actions and variant decreasing of the prophecy variable are introduced. They are necessary to prove synchronizability and well formedness expressed as invariants. The refinement of the synchronous models in an asynchronous model eases the proof process.

At the last refinement, realizability is proved thanks to invariants preservation and to the inductive proof process handled by Event-B using the Rodin platform.

Next sections sketches this development. For each refinement step, we introduce the relevant definitions, axioms and theorems needed to build the model.

## 4.2 The Root Model

It describes the notion of CP and introduces the definition of each operator at the CP level. Each introduced Event-B event corresponds to the formalisation of one operator defined in Sect. 3.1.

**Table 2.** An excerpt of the LTS\_CONTEXT.

<pre> <b>LTS_CONTEXT</b> <b>SETS</b> PEERS, MESSAGES, CP_STATES. <b>CONSTANTS</b> CPs_B, DC, ISeqF, NDC, ... <b>AXIOMS</b>   <b>axm1:</b> CPs_B <math>\subseteq</math> CP_STATES <math>\times</math> PEERS <math>\times</math> MESSAGES <math>\times</math> PEERS <math>\times</math> CP_STATES <math>\times</math> N     - <b>Deterministic CP definition DC</b>   <b>axm2_Cond1:</b> NDC <math>\subseteq</math> CPs_B   <b>axm3_Cond1:</b> <math>\forall Trans2, Trans1. (Trans1 \in CPs_B \wedge Trans2 \in CPs_B \wedge</math>     SOURCE_STATE(Trans1) = SOURCE_STATE(Trans2) <math>\wedge</math>     LABEL(Trans1) = LABEL(Trans2) <math>\wedge</math>     DESTINATION_STATE(Trans1) <math>\neq</math> DESTINATION_STATE(Trans2))     <math>\Rightarrow \{Trans1, Trans2\} \subseteq NDC</math>   <b>axm4_Cond1:</b> DC = CPs_B <math>\setminus</math> NDC     - <b>Independent sequence freeness definition ISeqF</b>   <b>axm5_Cond3:</b> ISeqF <math>\subseteq</math> CPs_B   <b>axm6_Cond3:</b> <math>\forall cp\_b. (cp\_b \in CPs_B \wedge</math>     (PEER_SOURCE(cp\_b) = LAST_SENDER_PEERS(SOURCE_STATE(cp\_b)) <math>\vee</math>     PEER_SOURCE(cp\_b) = LAST_RECEIVER_PEERS(SOURCE_STATE(cp\_b))))     <math>\Rightarrow \{cp\_b\} \subseteq ISeqF</math>   ...   End </pre>
--

**Required Properties for CPs (cf. Table 2).** Table 2 presents part of the Event-B context used at the abstract level. We introduce, using sets and constants, the whole basic definitions of messages, CP states, basic CPs, etc. A set of axioms is used to define the relevant properties of these definitions.

For example, in axiom *axm1*, a CP is defined as a set of transitions with a source and target state, a message and a source and target peers. *axm3\_Cond1* defines what a non deterministic CP is using the *NDC* set. This *NDC* set characterises all the non deterministic choices in a CP. Observe that axiom *axm4\_Cond1* defines the *DC* property in Definition 10 of Sect. 2.2.

**The Root Machine (cf. Table 4).** This model corresponds to the definition of the CP LTS. Each operator corresponds to one event and contributes to build a given CP represented in the state variable *BUILT\_CP* which shall define deterministic CP only (see invariant *inv1* in Table 3).

**Table 3.** An excerpt of the invariants of the LTS\_model.

<pre> <b>Invariants</b> <b>inv1:</b> BUILT_CP <math>\subseteq</math> DC   ... </pre>
--

The *Add\_Seq* event corresponds to the sequence operator of Definition 11 of Sect. 3.1. Its effect is to add a given basic *CP*, namely *Some\_cp\_b* to the currently built *CP* (union operation in action *act1*) and sets up the new final states in action *act3*. This event is triggered only if the relevant conditions identified in Sect. 3.1 holds (guards). For example, it is clearly stated that the independent sequence property *ISeqF* shall hold before adding another *CP* in sequence. This condition is given by guard *grd3* (see Table 4).

**Table 4.** An excerpt of the *LTS\_model*.

<pre> INITIALISATION <math>\triangleq</math> EVENTS   Add_Seq <math>\triangleq</math>     Any Some_cp_b     Where       grd1: Some_cp_b <math>\in</math> cps_b       grd2: MESSAGE(Some_cp_b) <math>\neq</math> End       grd3: Some_cp_b <math>\in</math> ISeqF       grd4: SOURCE_STATE(Some_cp_b) <math>\in</math> CP_Final_states       ...     Then       act1: BUILT_CP := BUILT_CP <math>\cup</math> {Some_cp_b}       act3: CP_Final_states := (CP_Final_states <math>\cup</math>         {DESTINATION_STATE(Some_cp_b)}) \         {SOURCE_STATE(Some_cp_b)}       ...     End   Add_Choice <math>\triangleq</math> ...   Add_Loop <math>\triangleq</math> ...   Add_End <math>\triangleq</math> ... End </pre>
--

Up to now, no proof related to realizability is performed. We have just stated that all the built *CPs* are deterministic (they belong to the *DC* set of *CPs* which represent a condition for the realizability property of Definition 10 in Sect. 2.2.

### 4.3 First Refinement: Synchronous Model

The objective of the first refinement is to build the synchronous projection corresponding to Definition 4. Here again, before building this projection, some property definitions are required, in particular for equivalence ( $\equiv$ ), denoted *EQUIV* in Event\_B models.

**Required Properties for Synchronous Projection (cf. Table 5).** The definition of the state-transitions system corresponding to the synchronous projection is given by the set *CPs\_SYNC\_B* defined by axiom *axm1* of Table 5. Actions (send ! and receive ?) are introduced. Then, two other important axioms, namely *axm1.a* and *axm1.a1*, are given to define the equivalence between a *CP* and its synchronous projection. The *EQUIV* relation is introduced. It characterises the set of *CPs* that are equivalent to their synchronous projection. *axm1.a1* formalises Definition 7 of Sect. 2.2.

**Table 5.** An excerpt of the `LTS_SYNC_CONTEXT`.

<p><b>LTS_SYNC_CONTEXT, EXTENDS LTS_CONTEXT</b>  <b>SETS ACTIONS. CONSTANTS</b> <code>CPs_B</code>, <b>EQUIV</b>, ...  <b>AXIOMS</b>  <b>axm1:</b> <math>CPs\_SYNC\_B \subseteq CP\_STATES \times ACTIONS \times MESSAGES \times PEERS \times</math>  <math>PEERS \times ACTIONS \times MESSAGES \times CP\_STATES \times N</math>  – <b>Equivalence of CP and Synchronous projection</b>  <b>axm_1.a:</b> <math>EQUIV \in CPs\_B \Rightarrow CPs\_SYNC\_B</math>  <b>axm_1.a1:</b> <math>EQUIV = \{ Trans \mapsto S\_Trans \mid Trans \in CPs\_B \wedge S\_Trans \in CPs\_SYNC\_B \wedge</math>  <math>SOURCE\_STATE(Trans) = S\_SOURCE\_STATE(S\_Trans) \wedge</math>  <math>DESTINATION\_STATE(Trans) = S\_DESTINATION\_STATE(S\_Trans) \wedge</math>  <math>PEER\_SOURCE(Trans) = S\_PEER\_SOURCE(S\_Trans) \wedge</math>  <math>PEER\_DESTINATION(Trans) = S\_PEER\_DESTINATION(S\_Trans) \wedge</math>  <math>MESSAGE(Trans) = S\_MESSAGE(S\_Trans) \wedge</math>  <math>INDEX(Trans) = S\_INDEX(S\_Trans) \}</math>  ...  <b>End</b></p>
---

**Table 6.** An excerpt of the invariants of the `LTS_Synchronous_model`.

<p><b>Invariants</b>  <b>inv1:</b> <math>BUILT\_SYNC \subseteq CPs\_SYNC\_B</math>  <b>inv_1.a:</b> <math>\forall Trans. \exists S\_Trans. (Trans \in BUILT\_CP \wedge S\_Trans \in BUILT\_SYNC \wedge</math>  <math>BUILT\_CP \neq \emptyset) \Rightarrow Trans \mapsto S\_Trans \in EQUIV</math></p>
--

**The Synchronous Projection (cf. Table 7).** The first refinement introduces the synchronous projection of the `BUILT_CP` defined by variable `BUILT_SYNC` in Table 7. Table 6 introduces through invariant `inv_1.a`. The equivalence ( $\equiv$ ) property corresponding to Condition 2.a in Eq. 2. The invariant requires equivalence between a CP and its synchronous projection. Invariant `inv2` of Table 6 describes the equivalence property using the `EQUIV` relation

**Table 7.** An excerpt of the `LTS_Synchronous_model`.

<p><b>INITIALISATION</b>  ...  <b>EVENTS</b>  <math>Add\_Seq \text{ Refines } Add\_Seq \triangleq</math>  <b>Any</b>  <math>S\_Some\_cp\_b, Some\_cp\_sync\_b</math>  <b>Where</b>  <b>grd1:</b> <math>Some\_cp\_sync\_b \in cps\_sync\_b</math>  <b>grd3:</b> <math>S\_SOURCE\_STATE(Some\_cp\_sync\_b) \in CP\_Final\_states</math>  <b>grd4:</b> <math>S\_Some\_cp\_b \in ISeq</math>  <b>grd8:</b> <math>MESSAGE(S\_Some\_cp\_b) \neq End</math>  <b>grd9:</b> <math>MESSAGE(S\_Some\_cp\_b) = S\_MESSAGE(Some\_cp\_sync\_b)</math>  ...  <b>With</b> <math>Some\_cp\_b: Some\_cp\_b = S\_Some\_cp\_b</math>  <b>Then</b>  <b>act1:</b> <math>BUILT\_CP := BUILT\_CP \cup \{S\_Some\_cp\_b\}</math>  <b>act2:</b> <math>BUILT\_SYNC := BUILT\_SYNC \cup \{Some\_cp\_sync\_b\}</math>  ...  <b>End</b></p>
---



defined in the context of Table 5. So, one part of the realizability property (i.e.  $CP \equiv Sys_{sync}$ ) of Definition 10 is already proved at this refinement level. The event *Add\_Seq* or sequence operator (Table 7) refines the same event of the root model. It introduces the *BUILT\_SYNC* set corresponding to the synchronous projection as given in Definition 4. Here, again, the *Add\_Seq* applies only if the conditions in the guards hold. The *With* clause provides a witness to glue *Some\_cp\_b* CP with its synchronous version.

#### 4.4 Second Refinement: Asynchronous Model

The second refinement introduces the asynchronous projection with sending and receiving peers actions. Well formedness and synchronizability remain to be proved in order to complete realizability preservation (Table 8).

**Table 8.** An excerpt of the LTS\_ASYNC\_CONTEXT.

<pre> CONTEXT LTS_ASYNC_CONTEXT EXTENDS LTS_SYNC_CONTEXT SETS A_STATES, ... CONSTANTS CPs_ASYNC_B, SYNCHRONISABILITY, WF, ... AXIOMS   axm1: <math>CPs\_ASYNC\_B \in (A\_STATES \times ETIQ \times \mathbb{N}) \dashv\vdash A\_STATES</math>     - Synchronisability property   axm1.b: SYNCHRONISABILITY <math>\in CPs\_SYNC\_B \mapsto R\_TRACE\_B</math>   axm1.b1: SYNCHRONISABILITY = <math>\{S\_Trans \mapsto R\_Trans \mid S\_Trans \in CPs\_SYNC\_B \wedge</math>     <math>R\_Trans \in R\_TRACE\_B \wedge S\_INDEX(S\_Trans) = R\_INDEX(R\_Trans) \wedge</math>     <math>S\_SOURCE\_STATE(S\_Trans) = R\_SOURCE\_STATE(R\_Trans) \wedge</math>     <math>S\_PEER\_SOURCE(S\_Trans) = R\_PEER\_SOURCE(R\_Trans) \wedge</math>     <math>S\_MESSAGE(S\_Trans) = R\_MESSAGE(R\_Trans) \wedge</math>     <math>S\_PEER\_DESTINATION(S\_Trans) = R\_PEER\_DESTINATION(R\_Trans) \wedge</math>     <math>S\_DESTINATION\_STATE(S\_Trans) = R\_DESTINATION\_STATE(R\_Trans)\}</math>     - Well formedness property   axm1.c: WF <math>\in A\_TRACES \rightarrow QUEUE</math>   axm1.c1: <math>\forall A\_TR, queue \cdot (A\_TR \in A\_TRACES \wedge queue \in QUEUE \wedge queue = \emptyset)</math>     <math>\Rightarrow A\_TR \mapsto queue \in WF</math>     ... End                 </pre>
---

**The Asynchronous Projection (cf. Tables 10 and 11).** The invariants associated to this model are presented in Table 9. In particular, the properties of synchronizability, expressed in invariant *axm1.b* used in Definition 10 ( $Sync(Sys_{sync}, Sys_{async})$ ), and of well formedness, expressed in invariant *axm1.c* used in Definition 10 ( $WF(Sys_{async})$ ) are introduced in the invariant of this refinement level. These two properties complete the proof of realizability.

At these level, each event corresponding to a composition operator is refined by three events: one to handle sending of messages (*Add\_Seq\_send*) on Table 10, one for receiving messages (*Add\_Seq\_receive*) and a third one (*Add\_Seq\_send\_receive*) on Table 11 refining the abstract *Add\_seq* event.

Tables 10 and 11 define these events. Sending and receiving events are interleaved in an asynchronous manner. Once a pair of send and receive events

**Table 9.** An excerpt of the invariants of the LTS<sub>Asynchronous\_model</sub>.

<b>Invariants</b> <b>inv1</b> $BUILT\_SYNC \subseteq CP\_SYNC\_B$ <b>inv2</b> $REDUCED\_TRACE \subseteq R\_TRACE\_B$ <b>inv3</b> $A\_TRACE \subseteq A\_TRACES$ <b>inv1.b</b> $\forall S\_Trans. \exists R\_Trans. (S\_Trans \in BUILT\_SYNC \wedge R\_Trans \in REDUCED\_TRACE) \Rightarrow$ $S\_Trans \mapsto R\_Trans \in SYNCHRONISABILITY$ <b>inv1.c</b> $\forall A\_Trans. (A\_Trans \in A\_TRACES \wedge MESSAGE>Last\_cp\_trans) = End \wedge$ $A\_TRACE \neq \emptyset \Rightarrow A\_Trans \mapsto queue \in WF$ <b>inv6</b> $BUILT\_ASYNC \subseteq CP\_ASYNC\_B$ ...
---

**Table 10.** An excerpt of the LTS<sub>Asynchronous\_model</sub>.

<b>Event</b> $Add\_Seq\_Send \triangleq$ <b>Any</b> $send, lts\_s, lts\_d, msg, index$ <b>Where</b> <b>grd1:</b> $\exists send\_st\_src, send\_st\_dest. ((lts\_s \mapsto send\_st\_src) \in A\_GS \wedge ((send\_st\_src \mapsto$ $(Send \mapsto msg \mapsto lts\_d) \mapsto index) \mapsto send\_st\_dest) \in CPs\_ASYNC\_B \wedge \dots$ ... <b>Then</b> <b>act1:</b> $A\_TRACE := A\_TRACE \cup \{Reduces\_Trace\_states \mapsto St\_Num \mapsto$ $Send \mapsto lts\_s \mapsto msg \mapsto lts\_d \mapsto Reduces\_Trace\_states \mapsto$ $(St\_Num + 1) \mapsto A\_Trace.index\}$ <b>act2:</b> $queue, back := queue \cup \{lts\_d \mapsto msg \mapsto back\}, back + 1$ <b>act3:</b> $A\_GS := A\_Next\_States(\{send\} \mapsto A\_GS \mapsto queue)$ ... <b>End</b>
---

has been triggered, the event *Add\_Seq\_send\_receive* records that the emission-reception is completed. This event increases the number of received messages (action *act5*). Traces are updated accordingly by the events, they are used for proving the invariants.

#### 4.5 Instantiation and Axiom Validation

To illustrate our approach, we have instantiated our model on a toy example corresponding to the CP depicted on Fig. 1. The labels of the transitions of the form  $m^p \mapsto p'$  denote a message  $m$  sent by peer  $p$  to the peer  $p'$ .

**Fig. 1.** Four messages exchanges in sequence for a electronic commerce system

The whole Event-B model has been instantiated. The context of Table 12 shows the instantiation of the model for the CP of Fig. 1. It also shows that the axioms

**Table 11.** An excerpt of the LTS<sub>Asynchronous</sub> model.

<p><b>Event</b> <i>Add_Seq_Receive</i> <math>\triangleq</math></p> <p><b>Any</b>  <i>send, receive, lts_s, lts_d, msg, index</i></p> <p><b>Where</b></p> <p><b>grd1:</b> <math>queue \neq \emptyset \wedge lts_d \mapsto msg \mapsto front \in queue</math></p> <p><b>grd2:</b> <math>\exists receive\_st\_src, receive\_st\_dest. ((lts_d \mapsto receive\_st\_src) \in A\_GS) \wedge ((receive\_st\_src \mapsto (Receive \mapsto msg \mapsto lts_s) \mapsto index) \mapsto receive\_st\_dest) \in CPs\_ASYNc\_B \wedge \dots</math></p> <p><b>Then</b></p> <p><b>act1:</b> <math>A\_TRACE := A\_TRACE \cup \{Reduces\_Trace\_states \mapsto St\_Num \mapsto Receive \mapsto lts\_s \mapsto msg \mapsto lts\_d \mapsto Reduces\_Trace\_states \mapsto (St\_Num + 1) \mapsto A\_Trace\_index\}</math></p> <p><b>act2:</b> <math>queue := queue \setminus \{lts\_d \mapsto msg \mapsto front\}</math></p> <p><b>End</b></p> <p><b>Event</b> <i>Add_Seq_Send - Receive Refines Add_Seq</i> <math>\triangleq</math></p> <p><b>Any</b>  <i>A_Some_cp_b, A_Some_cp_sync_b, Send_cp_async_b, Receive_cp_async_b, R_trace_b</i></p> <p><b>Where</b></p> <p><b>grd1:</b> <math>A\_MESSAGE(Send\_cp\_async\_b) = A\_MESSAGE(Receive\_cp\_async\_b)</math></p> <p><b>grd2:</b> <math>ACTION(Receive\_cp\_async\_b) = Receive \wedge ACTION(Send\_cp\_async\_b) = Send</math></p> <p><b>grd3:</b> <math>A\_Some\_cp\_b \in ISeq</math></p> <p><b>grd4:</b> <math>MESSAGE(A\_Some\_cp\_b) = A\_MESSAGE(Send\_cp\_async\_b)</math></p> <p><b>With</b> <math>S\_Some\_cp\_b : S\_Some\_cp\_b = A\_Some\_cp\_b,</math>  <math>Some\_cp\_sync\_b : Some\_cp\_sync\_b = A\_Some\_cp\_sync\_b</math></p> <p><b>Then</b></p> <p><b>act1:</b> <math>BUILT\_CP := BUILT\_CP \cup \{A\_Some\_cp\_b\}</math></p> <p><b>act2:</b> <math>BUILT\_SYNC := BUILT\_SYNC \cup \{A\_Some\_cp\_sync\_b\}</math></p> <p><b>act3:</b> <math>BUILT\_ASYNc := BUILT\_ASYNc \cup \{Send\_cp\_async\_b\} \cup \{Receive\_cp\_async\_b\}</math></p> <p><b>act4:</b> <math>REDUCED\_TRACE := REDUCED\_TRACE \cup \{R\_trace\_b\}</math></p> <p><b>End</b></p> <p><b>End</b></p>
---

defined in the model are inhabited. The ProB [15] model checker associated to Event-B on the Rodin platform has been used for automatic validation.

Other case studies borrowed from the research community dealing with realizability have been used to instantiate our model. These case studies use the whole composition operators we defined.

## 4.6 Assessment

Table 13 gives the results of our experiments. One can observe that all the proof obligations (POs) have been proved. A large amount of these POs has been proved automatically using the different provers associated to the Rodin platform. Interactive proofs of POs required to combine some interactive deduction rules and the automatic provers of Rodin. Few steps were required in most of the cases, and a maximum of 10 steps was reached.

**Table 12.** An excerpt of the LTS\_CONTEXT.instantiation.

<b>LTS_CONTEXT.instantiation</b>	<b>EXTENDS</b> LTS_CONTEXT
<b>CONSTANTS</b> s0, s1, s2, s3, s4, s5, Connect, Buy, Contact, Request_BBN, End, Buyer, ...	
<b>AXIOMS</b>	
axm1: partition(PEERS, {Buyer}, {e_shop}, {Bank}, {Pend})	
axm2: partition(MESSAGES, {Connect}, {Buy}, {Contact}, {Request_BBN}, {End})	
axm3: partition(CP_STATES, {s0}, {s1}, {s2}, {s3}, {s4}, {s5})	
axm4: CPs_B = {s0 $\mapsto$ Buyer $\mapsto$ Connect $\mapsto$ e_shop $\mapsto$ s1 $\mapsto$ 1, ...}	
axm5: CPs_SYNC_B = {s0 $\mapsto$ Send $\mapsto$ Connect $\mapsto$ e_shop $\mapsto$ Buyer $\mapsto$ Receive $\mapsto$ ...}	
axm6: partition(A_STATES, {B_s0}, {B_s1}, {B_s2}, {B_s3}, {e_s0}, {e_s1}, ...)	
axm7: CPs_ASYNC_B = {{{(B_s0 $\mapsto$ (Send $\mapsto$ Connect $\mapsto$ e_shop) $\mapsto$ 1) $\mapsto$ B_s1}, ...}}	
axm8: A_TRACES = {s $\mapsto$ 0 $\mapsto$ Send $\mapsto$ Buyer $\mapsto$ Connect $\mapsto$ e_shop $\mapsto$ s $\mapsto$ 1 $\mapsto$ 1, ...}	
axm9: R_TRACE_B = {s0 $\mapsto$ Buyer $\mapsto$ Connect $\mapsto$ e_shop $\mapsto$ s1 $\mapsto$ 1, ...}	
axm10: S_Next_States = {{{(B_s0 $\mapsto$ (Send $\mapsto$ Connect $\mapsto$ e_shop) $\mapsto$ 1) $\mapsto$ B_s1}} $\mapsto$ {{{(Buyer $\mapsto$ B_s0), (e_shop $\mapsto$ e_s0), (Bank $\mapsto$ Bk_s0)}} $\mapsto$ {{{(Buyer $\mapsto$ B_s1), (e_shop $\mapsto$ e_s0), (Bank $\mapsto$ Bk_s0)}, ...}}	
axm11: A_Next_States = {{{(B_s0 $\mapsto$ (Send $\mapsto$ Connect $\mapsto$ e_shop) $\mapsto$ 1) $\mapsto$ B_s1}} $\mapsto$ {{{(Buyer $\mapsto$ B_s0), (e_shop $\mapsto$ e_s0), (Bank $\mapsto$ Bk_s0)}} $\mapsto$ $\emptyset$ $\mapsto$ {{{(Buyer $\mapsto$ B_s1), (e_shop $\mapsto$ e_s0), (Bank $\mapsto$ Bk_s0)}, ...}}	
...	
<b>END</b>	

**Table 13.** RODIN proofs statistics

Event-B Model	Interactive proofs	Automatic proofs	Proof obligations
Abstract context	06 (100%)	0 (0%)	06 (100%)
Synchronous context	02 (100%)	0 (0%)	02 (100%)
Asynchronous context	01 (33.33%)	02 (66.67%)	03 (100%)
Abstract model	28 (58.33%)	20 (41.67%)	48 (100%)
Synchronous model	39 (39%)	61 (61%)	100 (100%)
Asynchronous model	73 (38.83%)	115 (61.17%)	188 (100%)
Total	148 (100%)	198 (100%)	347 (100%)

## 5 Related Work

Several approaches addressed choreography realizability. In [10], the authors identify three principles for global descriptions under which a sound and complete end-point projection is defined. If these rules are respected, the projection will behave as specified in the choreography. This approach is applied to BPMN 2.0 choreographies [18]. [20] propose to modify their choreography language to include new constructs (choice and loop). During projection, particular communication is added to enforce the peers to respect the choreography specification. In [12], the authors propose a Petri Net-based formalism for choreographies and algorithms to check realizability and local enforceability. A choreography is locally enforceable if interacting peers are able to satisfy a subset of the requirements of the choreography. To ensure this, some message exchanges in the distributed system are disabled. In [21], the authors propose automated techniques to check the realizability of collaboration diagrams for different communication models.

Beyond advocating a solution for enforcing realizability, our contribution differs from these approaches as follows. We focus on asynchronous communication and choreographies involving loops. Our approach is non-intrusive; we do not add any constraints on the choreography language or specification, and the designer neither has to modify the original choreography specification, nor the peer models. We considerably reduce the verification complexity since there is no need to re-build the distributed system by composition of peers to check the realizability. Instead of that, we rely on a correct-by-construction approach based on sufficient conditions for realizability at the CP level. The technique we rely on here shares some similarities with counterexample-guided abstraction refinement (CEGAR) [11]. In CEGAR, an abstract system is analyzed for temporal logic properties. If a property holds, the abstraction mechanism guarantees that the property also holds in the concrete design. If the property does not hold, the reason may be a too coarse approximation by the abstraction. In this case, the counterexample generated by the model checker, is used to refine the system to a finer abstraction and this process is iterated.

To the best of our knowledge, our approach is the first correct-by-construction method which enables the designer to specify realizable CP avoiding behavioural errors in the distributed systems. By doing so, we propose an a priori verification method where the problems of state explosion and scalability are discarded. Other proof based techniques than Event-B like Coq [3] or Isabelle [17] could have been used after defining the refinement operation. Our approach extensively uses built-in refinement operation and inductive proof schemes of Event-B.

## 6 Conclusion

This paper presents an a priori approach to build realizable CPs based on a correct-by-construction method. A language allowing to incrementally build complex realizable CPs from a set of basic realizable ones is defined. It offers a set of composition operators preserving realizability. Our proposal is proved to be sound and correct using the proof and refinement based formal method Event-B. Thanks to the use of arbitrary sets of values for parameters in our Event-B models, our approach is scalable. Moreover, we have validated this model using several case studies. According to [4], this instantiation process is defined either using model checking to animate and test the CPs associated to each case study; or by explicitly supplying a witness to each parameter of the events in the Event-B model to build the CP associated to the case study.

As a short term perspective, we aim at extending our model with an operator enabling to compose entire CPs instead of requiring incremental composition of basic  $CP_b$ . Furthermore, we intend to define a set of patterns for realizable CPs and studying the completeness of our language in order to identify the class of real-world asynchronously communicating systems that can be specified. Last, we aim at providing the designers with an engine for automatic instantiation of realizable CPs.

## References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. *Theor. Comput. Sci.* **82**(2), 253–284 (1991)
2. Abrial, J.: *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge (2010)
3. Athalye, A.: CoqIOA: A formalization of IO automata in the Coq proof assistant, vol. 1019, pp. 1–53 (1995)
4. Babin, G., Ait-Ameur, Y., Pantel, M.: Correct instantiation of a system reconfiguration pattern: a proof and refinement-based approach. In: *Proceedings of HASE 2016*, pp. 31–38. IEEE Computer Society (2016)
5. Basu, S., Bultan, T., Ouederni, M.: Deciding choreography realizability. In: *Proceedings of POPL 2012*, pp. 191–202. ACM (2012)
6. Benyagoub, S., Ouederni, M., Ait-Ameur, Y.: Towards correct evolution of conversation protocols. In: *Proceedings of VECOS 2016*. CEUR Workshop Proceedings, vol. 1689, pp. 193–201. CEUR-WS.org (2016)
7. Benyagoub, S., Ouederni, M., Singh, N.K., Ait-Ameur, Y.: Correct-by-construction evolution of realisable conversation protocols. In: Bellatreche, L., Pastor, Ó., Almendros Jiménez, J.M., Ait-Ameur, Y. (eds.) *MEDI 2016*. LNCS, vol. 9893, pp. 260–273. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-45547-1\\_21](https://doi.org/10.1007/978-3-319-45547-1_21)
8. Brand, D., Zafropulo, P.: On communicating finite-state machines. *J. ACM* **30**(2), 323–342 (1983)
9. Bultan, T.: Modeling interactions of web software. In: *Proceedings of IEEE WWV 2006*, pp. 45–52 (2006)
10. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: De Nicola, R. (ed.) *ESOP 2007*. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-71316-6\\_2](https://doi.org/10.1007/978-3-540-71316-6_2)
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](https://doi.org/10.1007/10722167_15)
12. Decker, G., Weske, M.: Local enforceability in interaction petri nets. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) *BPM 2007*. LNCS, vol. 4714, pp. 305–319. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-75183-0\\_22](https://doi.org/10.1007/978-3-540-75183-0_22)
13. Farah, Z., Ait-Ameur, Y., Ouederni, M., Tari, K.: A correct-by-construction model for asynchronously communicating systems. *Int. J. STTT* **19**, 1–21 (2016)
14. Hopcroft, J.E., Ullman, J.D.: *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, Boston (1979)
15. Bendisposto, J., Clark, J., Dobrikov, I., Karner, P., Krings, S., Ladenberger, L., Leuschel, M., Plagge, D.: Prob 2.0 tutorial. In: *Proceedings of 4th Rodin User and Developer Workshop, TUCS* (2013)
16. Milner, R.: *Communication and Concurrency*. Prentice-Hall Inc., Upper Saddle River (1989)
17. Müller, O., Nipkow, T.: Combining model checking and deduction for I/O-automata. In: Brinksma, E., Cleaveland, W.R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) *TACAS 1995*. LNCS, vol. 1019, pp. 1–16. Springer, Heidelberg (1995). [https://doi.org/10.1007/3-540-60630-0\\_1](https://doi.org/10.1007/3-540-60630-0_1)
18. OMG: *Business Process Model and Notation (BPMN) - Version 2.0* (2011)
19. Project RODIN: *Rigorous open development environment for complex systems* (2004). <http://rodin-b-sharp.sourceforge.net/>

20. Qiu, Z., Zhao, X., Cai, C., Yang, H.: Towards the theoretical foundation of choreography. In: Proceedings of WWW 2007. ACM Press (2007)
21. Salaün, G., Bultan, T.: Realizability of choreographies using process algebra encodings. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 167–182. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-00255-7\\_12](https://doi.org/10.1007/978-3-642-00255-7_12)



# Formal Assurance for Cooperative Intelligent Autonomous Agents

Siddhartha Bhattacharyya<sup>1</sup>(✉), Thomas C. Eskridge<sup>1</sup>, Natasha A. Neogi<sup>2</sup>,  
Marco Carvalho<sup>1</sup>, and Milton Stafford<sup>1</sup>

<sup>1</sup> School of Computing, Florida Institute of Technology, Melbourne, FL, USA  
{sbhattacharyya,teskridge,mcarvalho,mstafford2012}@fit.edu

<sup>2</sup> NASA Langley Research Center, Hampton, VA, USA  
natasha.a.neogi@nasa.gov

**Abstract.** Developing trust in intelligent agents requires understanding the full capabilities of the agent, including the boundaries beyond which the agent is not designed to operate. This paper focuses on applying formal verification methods to identify these boundary conditions in order to ensure the proper design for the effective operation of the human-agent team. The approach involves creating an executable specification of the human-machine interaction in a cognitive architecture, which incorporates the expression of learning behavior. The model is then translated into a formal language, where verification and validation activities can occur in an automated fashion. We illustrate our approach through the design of an intelligent copilot that teams with a human in a takeoff operation, while a contingency scenario involving an engine-out is potentially executed. The formal verification and counterexample generation enables increased confidence in the designed procedures and behavior of the intelligent copilot system.

**Keywords:** Formal verification · Intelligent agents  
Human-machine teams

## 1 Introduction

Autonomous systems are increasingly being designed to collaborate with humans to accomplish safety critical tasks. These cooperative agents are typically designed in modeling paradigms that emphasize human-machine interactions, efficiency, learning and performance improvement. However, operators mitigate safety and operational risks in an adaptive manner on a frequent basis, and the system often relies on this mitigation.

Our research focuses on the development of assurance for cooperative agents that can execute tasks autonomously, work with environmental variations and improve their own performance and the performance of the cooperative system overall. We translate the knowledge representations used by the cooperative agent into a formally verifiable representation to ensure that any modification



to the agent behavior learned during operation will conform to the individual and system-level requirements specified during design and deployment. In this paper, we discuss an initial version of our automated translator and how this translation is achieved through a model transformation from an agent knowledge representation into a formally verifiable framework. Our technique provides a proof of concept for how the formal verification of autonomous agents could be utilized for safety critical applications.

**Contribution.** Frameworks for specifying autonomous cooperative agents usually lack the inherent capability for rigorous analysis, such as the formal verification and validation of safety critical properties. Our approach is novel in that it allows for the direct use of formal tools to verify safety properties of an autonomous agent designed in an existing cognitive architecture (Fig. 1). Thus, the formal verification process can be integrated into the design process of the agent. We believe this to be a unique contribution, which is, to the best of our knowledge, not currently otherwise utilized between other common intelligent agent frameworks (e.g., ACT-R, EPIC, PRS etc.) and formal methods (e.g., SMV, PVS, IC3 etc.).

We next review related work in Sect. 2. A target cognitive architecture and formal verification framework is selected in Sect. 3. We outline an example human-cooperative agent scenario involving an aircraft engine-out case study in Sect. 4. A formal description of the automated translation process between the cognitive architecture and verification formalism is provided in Sect. 5, along with implementation particulars. Section 6 discusses the results of the verification and validation efforts on the case study, and modifications made to the agent as a result. Conclusions and future work are detailed in Sect. 7.

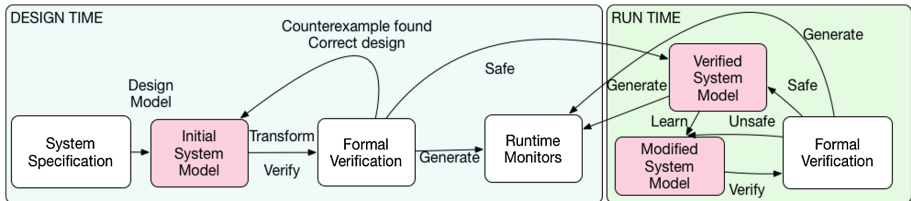


Fig. 1. System modeling and verification process

## 2 Related Work

Rule-based reasoning systems, or production systems, have been a popular method in Artificial Intelligence for producing intelligent behavior that is understandable to the program operator. Common rule based reasoning systems include the General Problem Solver (GPS) [1], the MYCIN knowledge based inference system [2], the Adaptive Control of Thought-Rational Theory (ACT-R) [3] and the Soar cognitive architecture [4]. Rule-based reasoning frameworks facilitate adaptation in two ways: creating new rules and modifying

rule order. Creating new rules can occur by identifying a new set of conditions under which an action should occur (called new rule learning) or by combining a number of rules together to form a single rule that makes execution of the set more efficient (called chunking) [4]. Rule order can be modified through reinforcement learning, which optimizes rule order based on a criteria such as minimum number of rule firings to reach a goal or minimizing overall time to execute [5]. However, none of these modeling frameworks, which support learning behaviors, possess a formal semantics, and do not naturally support formal verification. Research efforts [6] extend ACT-R based on a discrete event simulation formalism, however, this does not encompass systems with continuous dynamics.

Research efforts in the area of verification of adaptive architectures for autonomy include work done on Rainbow CMU [7], which focuses on dynamic system adaptation via architecture models, along with potential formal guarantees. Research conducted by Wen [8] focuses on constraining the inputs to learning systems in order to synthesize systems that are correct by construction. Sharifloo and Spoletini [9] describes the use of lightweight formal verification methods for runtime assurance when a system is updated with a new component. In [10], Curzon discusses the development of a formal user model to develop a generic approach to cognitive architecture, but does not integrate this work with an existing architecture. Additionally, the formal verification of an autonomous system is discussed by O'Connor [11], and is based on the design of a mission controller with timed automata; however this does not enable the modeling of cognitive components. None of these efforts provide support for rigorously analyzing existing cognitive decision procedures implemented through an existing cognitive architecture.

NASA has developed PPlan Execution Interchange Language (PLEXIL) [12], which has been successfully deployed for several autonomy applications. PLEXIL's operational semantics has been formally specified in the Prototype Verification System (PVS) [13] and properties such as determinism and compositionality were mechanically verified in PVS [14]. Architecturally, the executable semantics in PLEXIL are specified in the rewrite logic engine Maude [15] for formal verification of the plans. Several efforts have focused on the use and analysis of PLEXIL, such as Strauss's efforts in analysing execution semantics in Haskell [16]. Balasubramanian et al. have developed Polyglot, a Statecharts analysis framework for PLEXIL, and are investigating formal analysis of a Statechart-based semantics of PLEXIL [17]. Verdejo and Martí-Oliet [18] have investigated the development of tools from the operational semantics specified in Maude. In our approach we utilize a similar paradigm of having the plans represented as rules in Soar, which are translated into Uppaal for verification. The additional benefit provided by our approach is that Soar has learning capabilities embedded within its architecture. The authors have also developed a navigation agent for drones in enclosed areas via the proposed methodology for direction selection [19]. In this paper, we extend this previous work by designing and implementing an autonomous agent that performs decision-making processes in the context of human teamwork. We next discuss the modeling formalisms used to support this work.

### 3 Modeling and Formal Verification

The goal of the research is to be able to model and ultimately design human and automated agent interactions, thereby enabling a shifting locus of control from human agents to automation in a variety of domains. This will entail automated agents performing complex safety-critical tasks in conjunction with human supervisors, which may result in possibly inimical emergent properties. In order to avoid this, we wish to map cognitive task models or operations into a formal language, where safety and liveness invariants can be verified. We considered several cognitive frameworks in order to model human interactions, as well as several formalisms for verification purposes. We detail the process of choosing a framework and formalism in Subsects. 3.1 and 3.2.

#### 3.1 Cognitive Architectures and Frameworks: Soar

Cognitive architectures, which are often candidates for a general theory of cognition, can be regarded as architectures for the expression of intelligent behavior. We surveyed several rule based reasoning systems as candidates for modeling human-automation interactions [1–4]. Soar was selected based on its ability to encompass multiple memory constructs (e.g., semantic, episodic, etc.) and learning mechanisms (e.g., reinforcement, chunking etc.). Furthermore, Soar production rules are expressed in first order logic, which makes them amenable to verification. Finally, Soar takes the form of a programmable architecture with an embedded theory; this leads to the ability to execute Soar models on embedded system platforms, which enables the study of the design problem through the use of rapid prototyping and simulation.

**Production Rules Expressed in the Soar Representation.** Every Soar production rule starts with the symbol *sp*, which stands for Soar production. The remainder of the rule body is enclosed in braces. The body consists of the rule name, followed by one or more conditions expressed in first order logic, then the symbol  $\rightarrow$ , which is followed by one or more actions (also expressed in first order logic). In Soar, a state variable (expressed as  $\langle \text{variable} \rangle$ ) can have multiple features or attributes, where features or attributes are indicated by the symbol  $\hat{\phantom{x}}$ . An attribute can take on a value, which is stated in the string following the attribute. So, the Soar expression:  $(\langle s \rangle \hat{\text{superstate}} \text{ nil})$  means that the state variable  $s$  has a feature, called *superstate*, whose value is *nil*. An example Soar rule is:

$$sp\{proposeInitialize(state \langle s \rangle - \hat{\text{name}} \hat{\text{superstate}} \text{ nil}) \rightarrow (\langle s \rangle \hat{\text{operator}} \langle o \rangle)(\langle o \rangle \hat{\text{name}} \text{ initialize})\}$$

The Soar rule *proposeInitialize* has the condition where the state variable  $s$  has the attributes *name* (whose value is unassigned) and *superstate* (whose value is *nil*). The Soar feature *superstate* is an internal mechanism that Soar can use as part of its processing of goal-subgoal hierarchies. The condition where

the feature *superstate* holds the value *nil*, and there is no associated operator, indicates that Soar has just been invoked and no processing has been done yet. This subgoal hierarchy capability is not used in the example in Sect. 4, and therefore the superstate is only used to initialize the agent that is processing. So, in this case, the precondition is that no superstate exists and that there is no pre-existing name for the state  $\langle s \rangle$ . The right hand side (RHS) of the rule is the post condition or action, which indicates that given the LHS is true, an operator  $\langle o \rangle$  is associated with the state  $\langle s \rangle$  and that an attribute of the operator is its *name*, which has a value *initialize*.

Soar production rules commonly execute in pairs of *propose* and *apply* rules. The *propose* rule checks which Soar production rules are eligible to be executed, and the corresponding *apply* rule executes one of the eligible rules. In Sects. 4 and 5, we used the Soar production system framework to encode rules describing takeoff procedures for an automated copilot in a commercial aircraft. The Soar framework facilitated development of the automated copilot agent. Furthermore, the first order logic representation of the production ruleset facilitates its translation into an appropriate modeling formalism for formal verification, as detailed in the next subsection.

### 3.2 Formal Languages and Verification: Uppaal

In order to choose the correct platform to translate Soar models into for the purpose of verification, several formalisms such as NuSMV [20], Uppaal [21], PVS [22] and Z3 [23] were considered carefully. We chose Uppaal [21, 24, 25], due to its ability to model timing aspects that are critical for cyberphysical systems, as well as its ability to generate and visualize counterexamples. Uppaal models are represented by timed automata, and the compositionality enabled by the Uppaal formalism supports model checking over networked timed automata using temporal logics. This modeling paradigm allows the execution of requirements as temporal logic queries to exhaustively check the satisfaction of relevant safety properties. We next describe the timed automata formalism used by Uppaal.

**Mathematical Representation in Uppaal.** Uppaal uses timed automata [26], a subset of hybrid automata, as a modeling formalism. One of the essential requirements in the design of human-agent teams is to be able to model the time associated with the execution of operations or rules. A timed automaton is a finite automaton extended with a finite set of real-valued clocks. Clock or other relevant variable values can be used in guards on the transitions within the automaton. Based on the results of the guard evaluation, a transition may be enabled or disabled. Additionally, variables can be reset and implemented as invariants at a state. Modeling timed systems using a timed-automata approach is symbolic rather than explicit, and allows for the consideration of a finite subset of the infinite state space on-demand, i.e., using an equivalence relation that depends on the safety property and the timed automaton, which is referred to as the region automaton. There also exists a variety of tools to input and

analyze timed automata and extensions, including the model checkers Uppaal, and Kronos [27]. For the purpose of this paper, we represent timed automata formally as follows.

**Definition 1 *Timed Automaton (TA)*.** A timed automaton is a tuple  $(L, l_0, C, A, E, I)$ , where  $L$  is a set of locations;  $l_0 \in L$  is the initial location;  $C$  is the set of clocks;  $A$  is a set of actions, co-actions and the internal  $\tau$ -action;  $E \subseteq L \times A \times B(C) \times 2^C \times L$  is a set of edges between locations with an action, a guard and a set of clocks to be reset; and  $I : L \rightarrow B(C)$  assigns invariants to locations.

We define a clock valuation as a function  $u : C \rightarrow \mathbb{R}_{\geq 0}$  from the set of clocks to the non-negative reals. Let  $\mathbb{R}^C$  be the set of all clock valuations. Let  $u_0(x) = 0$  for all  $x \in C$ . If we consider guards and invariants as sets of clock valuations (with a slight relaxation of formalism), we can say  $u \in I(l)$  means that  $u$  satisfies  $I(l)$ . We can now define the semantics of a timed automaton as follows.

**Definition 2 *Timed Automaton (TA) Semantics*.** Let  $(L, l_0, C, A, E, I)$  be a timed automaton  $TA$ . The semantics of the  $TA$  is defined as a labelled transition system  $\langle S, s_0, \rightarrow \rangle$ , where  $S \subseteq L \times \mathbb{R}^C$  is the set of states,  $s_0 = (l_0, u_0)$  is the initial state, and  $\rightarrow \subseteq S \times \{\mathbb{R}_{\geq 0} \cup A\} \times S$  is the transition relation such that:

1.  $(l, u) \xrightarrow{d} (l, u + d)$  if  $\forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(l)$  and
2.  $(l, u) \xrightarrow{a} (l', u')$  if  $\exists e = (l, a, g, r, l') \in E$  such that  $u \in g, u = [r \mapsto 0]u$  and  $u' \in I(l')$ ,

where for  $d \in \mathbb{R}_{\geq 0}$ ,  $u + d$  maps each clock  $x$  in  $C$  to the value  $u(x) + d$ , and  $[r \mapsto 0]u$  denotes the clock valuation which maps each clock in  $r$  to 0 and agrees with  $u$  over  $C \setminus r$ .

Note that a guard  $g$  of a  $TA$  is a simple condition on the clocks that enable the transition (or, edge  $e$ ) from one location to another; the enabled transition is not taken unless the corresponding action  $a$  occurs. Similarly, the set of reset clocks  $r$  for the edge  $e$  specifies the clocks whose values are set to zero when the transition on the edge executes. Thus, a timed automata is a finite directed graph annotated with resets of, and conditions over, non-negative real valued clocks.

Timed automata can then be composed into a network of timed automata over a common set of clocks and actions, consisting of  $n$  timed automata  $TA_i = (L_i, l_{i0}, C, A, E_i, I_i)$ ,  $1 \leq i \leq n$ . This enables us to check reachability, safety and liveness properties, which are expressed in temporal logic expressions, over this network of timed automata. An execution of the  $TA$ , denoted by  $exec(TA)$  is the sequence of consecutive transitions, while the set of execution traces of the  $TA$  is denoted by  $traces(TA)$ . We next consider a simple flight example involving the interaction of human pilot with an autonomous copilot in a contingency situation where an engine becomes disabled during aircraft takeoff. We shall use this example to illustrate the process by which a Soar model is translated into Uppaal, and then we shall attempt to verify the design of the automated copilot model, with respect to simple safety and reachability properties.

## 4 Example Case Study: Engine Out Contingency During Takeoff

The example used to illustrate this technique was that of an engine-out contingency upon takeoff. Conventionally defining the term pilot flying (PF) as the agent designated as being responsible for primary flight controls in the aircraft (e.g., stick and surface inputs), and pilot not flying (PNF) as the agent not responsible for primary flight controls, has the Soar agent assuming the role of PNF. Thus, the Soar agent monitors procedure execution for off nominal or contingency situations, as well as performs secondary actuation tasks, similar to those performed by a copilot. However, it is important to note that the human pilot is always ultimately responsible for the overall safe execution of the flight [28].

For illustrative purposes, consider the scenario of a large cargo aircraft (such as a Boeing 737) during takeoff which experiences an engine failure, whereby the engine is delivering insufficient power after the aircraft brakes have been released, but before the aircraft takeoff has been successfully completed. Prior to takeoff, the speed  $V1$  is calculated, which is defined by the FAA as “the maximum speed in the takeoff at which the pilot must take the first action (e.g., apply brakes, reduce thrust, deploy speed brakes) to stop the airplane within the accelerate-stop distance” [29]. Thus,  $V1$  is a critical engine failure recognition speed, and can be used to determine whether or not the takeoff will continue, or result in a rejected takeoff (RTO).  $V1$  is dependent on factors such as aircraft weight, runway length, wing flap setting, engine thrust used and runway surface contamination. If the takeoff is aborted after the aircraft has reached  $V1$ , this will likely result in a runway overrun, that is, the aircraft will stop at a point in excess of the runway. Thus,  $V1$  is also seen as the speed beyond which the takeoff should continue: the engine failure is then handled as an airborne emergency.

A conventional takeoff, whereby humans fill the PF and PNF roles, proceeds as follows. Both pilots review any changes in the ATC clearance prior to initiating the Before Takeoff (BT) checklist. All Before Takeoff checklist items must be completed before the takeoff roll commences. Once the checklist is completed, the Takeoff procedure is performed as detailed in Fig. 2. It can be seen that there is a great deal of interplay between the PF and PNF, especially in terms of affirming tasks and settings through callouts. These callouts also serve to initiate the subsequent task in the procedure. Thus, any tasks that are delegated to an automated PNF, performing the copilot role, must mimic this annunciation structure, in order to preserve situation awareness in the cockpit, and foster teamwork in the human-automation team. In the case of an engine failure at a speed of less than  $V1$ , but above the lower threshold speed of 80 kts, the Contingency procedure shown in Fig. 3 is called from within the nominal Takeoff procedure.

These two operational procedures can be used to create a Soar production system, which models the behavior of the copilot, and can be executed, thereby creating an automated copilot function for takeoff. This process results in the creation of 15 Soar production rules. We now detail the process whereby the set of Soar production rules is translated into a network of timed automata, in Sect. 5.

Task/Initiation Cue	PF	PNF
<b>Takeoff</b>		
Aircraft in position on the active runway. BT checklist completed and cleared for takeoff.	Hold brakes. Advance power to takeoff N1/Engine Pressure Rating, per AFM.	Call, "Power set, instruments stabilized." Monitor engines and systems indicators.
PNF calls, "Power set, instruments stabilized."	Release brakes.	
<b>Takeoff Roll</b>		
Below 80 KIAS	Maintain directional control	Steady the control yoke with the right hand, (as applicable to aircraft type)
Positive airspeed indication		Call, "Airspeed alive"
PNF calls, "Airspeed alive."	Verify airspeed.	
At 80 KIAS		Verify 80 knots indicated on both PF and PNF airspeed indicators. Call, "80 knots cross-checked."
PNF calls, "80 knots cross-checked."	Move left hand from nose steering to control yoke and call, "My yoke". (as applicable to aircraft type)	
PF calls "My yoke". (as applicable to aircraft type)		Release control yoke. (as applicable to aircraft type)
At V1		Call, "V1."
PNF calls, "V1."	Move right hand to control yoke.	
At VR		Call, "Rotate."
PNF calls, "Rotate."	Rotate aircraft to pitch attitude per AFM.	

Fig. 2. Nominal takeoff procedure [30]

Task/Initiation Cue	PF	PNF
<b>Engine Out Rejected Takeoff</b>		
Engine Out Detected		The PNF closely monitors essential instruments during the takeoff roll and immediately announce abnormalities or any adverse condition significantly affecting safety of flight.. Call "Engine Fire", "Engine Failure" etc.
PNF calls, "Engine Failure" below V1 (and above 80 kts)	Call "Abandon" and take control of the aircraft.	
PF calls, "Abandon"	Close thrust levers and disengage simultaneously autothrottle.	Verify thrust levers close and autothrottle disengaged. Call out omitted action items.
Autothrottle disengaged	Verify automatic RTO braking (above 90 kts) or take maximum manual braking (below 90 kts) as required if deceleration is not adequate or if Autobrake Disarm light is illuminated.	Note the brakes on speed. Call "Autobrake Disarm" - Call out omitted action items.
PNF calls, "Autobrake disarm"	Raise speed brake lever.	Call "Speedbrakes Up" or Call "Speedbrakes Not Up"
PNF calls, "Speedbrakes Up"	Apply maximum reverse thrust consistent with runway conditions and continue maximum braking until certain airplane will stop on the runway.	Verify thrust reverser. Call out speed: "100 Kts, 80 kts, 60 kts. . ."
PNF calls, "10 Kts"	Stop aircraft on runway heading or consider turning into wind if the takeoff was rejected due to fire warning.	At alternating red and white runway lights call "900 meters" of runway remaining. At steady red lights call "300 meters" of runway remaining.
Aircraft stopped	Set Parking Brake	Select 1 lugs 40 when parking brake is set. Inform ATC including information on airplane position and alert if necessary the fire brigade

Fig. 3. Contingency procedure for engine out on takeoff [30]

## 5 Automated Translation from Cognitive Architecture to Formal Environment

The process of translation from Soar to Uppaal captures our understanding of the differences in the cognitive model and its formal representation. We have automated the Soar to Uppaal translation for a subset of Soar models. This includes translation of conditions in a production rule represented with variables, operators, disjunctions, conjunctions as well as for action items adding new elements to working memory, and creating preferences for two among twelve preferences. Further support needs to be added for multi-attribute rules, creating the remaining ten preferences and implementing translation of other action items such as mathematical operations.

Figure 4 shows the sequential operations the translator goes through, which are (1) lexical analysis, (2) semantic parsing, (3) symbolic and syntax analysis and (4) generating the Uppaal .xml file.

Given the grammar describing the Soar productions, Another Tool for Language Recognition (ANTLR) was used to parse Soar, resulting in a syntax tree for further translation. From this tree, symbols—local variables used in productions—are extracted to add to the Uppaal model. Each Soar production can then be mapped to one or more Uppaal actions, which must then fire in a sequential fashion; meanwhile each Soar syntactical element must be mapped to the corresponding Uppaal element. Once the parser is created, it parses the Soar file to generate the graphical tree for a Soar rule.

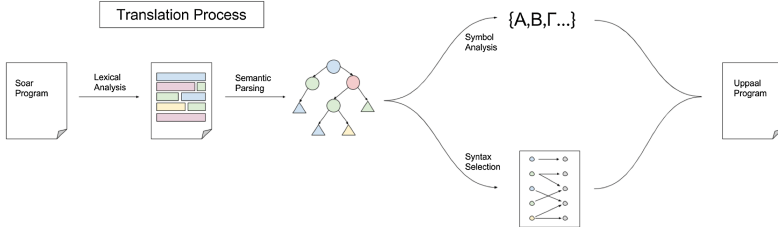


Fig. 4. Model transformation from Soar to Uppaal

In the Antlr grammar, the rule is parsed based on identifying if it is a *Soar rule* followed by if it is a *Soar production*. Once it is confirmed to be a Soar rule then the parser identifies if it is the LHS or RHS of the rule. If it is the LHS the *condition parameters* and the *variables* are identified. If it is the RHS of the rule, *action parameters* are identified along with the *variables*, *expression* and *preferences*. Once the trees are generated, the Soar rule is translated into a Timed Automaton (TA).

### 5.1 Automated Translation to Uppaal

In order to give a formal representation of the translation process, we consider the restricted subset of Soar production systems whose rules are represented using first order logic. We can then define a Soar production rule as a function of a finite set of variables  $V \in v_i, i = 1, 2, 3 \dots n$ , whose valuation  $val(V)$  represent the state of the system, along with a finite set of well formed formulae (WFF)  $\Phi = \{\phi_1, \phi_2, \dots \phi_m\}$ , representing the left hand side of the Soar production rule (e.g., the preconditions), and a finite set of WFF  $\Psi = \{\psi_1, \psi_2, \dots \psi_r\}$ , representing the actions embodied by the right hand side of the Soar production rule. We use the following formal definition for Soar production rules.

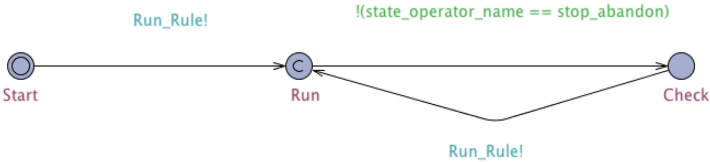
**Definition 3.** *An individual rule in cognitive model CM is represented as a tuple  $rname(V, Pre(\Phi), Post(\Psi))$  where there are  $i = 1 \dots n$  variables,  $m$  well formed formulae in the precondition  $\{\phi_1(v_j), \phi_2(v_k), \dots \phi_m(v_l)\}$ , and  $r$  well formed formulae in the postcondition  $\{\psi_1(v_s), \psi_2(v_u), \dots \psi_r(v_w)\}$ .*

Each WFF ( $\phi$  or  $\psi$ ) may depend on a subset of the variables in  $V$ , as well as constants. Preconditions and postconditions can be formed through the use of first order logical operators (e.g.,  $\vee, \wedge, \forall$  etc.) over WFF. The execution or firing of a production rule creates an observable change in the system state, which can be denoted by  $fire(rname)$ . The goal is to map the semantics of a Soar production rule onto the semantics of a Uppaal TA. For the translation to be correct, we wish to have the behavior of the cognitive model be equivalent to the behavior of the network of TAs, at least with respect to the properties being verified. Each Soar production rule generates a TA in Uppaal, and the set of all TAs compose into a networked TA that corresponds to the cognitive model



embodied by the Soar production rules. However, due to the nature of timing captured in Uppaal, we must also generate a scheduler (see Fig. 5), which forces a production rule and its corresponding TA to fire if one is available to do so. After any TA in the network has fired, the scheduler evaluates whether the goal state of the cognitive model has been reached. If this is not the case, the scheduler broadcasts the action `Run_Rule`, which causes an automata whose preconditions are enabled to fire. The `Run_Rule` action also allows the TA corresponding to the previously fired production rule to reset, rendering it immediately available to fire at the next evaluation of the TA network.

Specifically, each individual TA corresponding to a production rule has a *Start* state and a *Run* state, and a single clock  $x$ , whose value is given by  $u(x) = val(x)$ . Roughly speaking, the guard conditions for the TA correspond to the preconditions of the production rule. Similarly, the actions of the TA can be represented by the postconditions of the production rule. For example, at the start of the TA network execution, the guard condition for the initialization Soar production rule (Fig. 6) is the only rule that is true. This corresponds to the Soar representation where the guard condition, *superstate* is equal to *nil*, is true at initialization. Thus, the TA corresponding to the initialization rule executes when the scheduler sends out the `Run_Rule` broadcast shown in Fig. 5. During execution of the initialization rule, the values of variables on the RHS of the rule are updated, which changes guards for other rules to become true. After any TA executes, the scheduler transitions to the *Check* state on its own guard condition, which is a negation of the goal. If the goal is not met, the scheduler transitions from the *Check* to the *Run* state, and broadcasts the `Run_Rule` action to all TA, enabling further TA execution.



**Fig. 5.** Generic scheduler for timed automata derived from Soar rules

We briefly describe the algorithm as follows. The algorithm takes as its input a tuple  $rname(V, Pre(\{\phi_1, \phi_2, \dots, \phi_m\}), Post(\{\psi_1, \psi_2, \dots, \psi_r\}))$ , which is a rule from the Soar *CM*, and translates it to a timed automaton  $TA = (L, l_0, C, A, E, I)$ . The first line of the algorithm *requires* that all preconditions  $Pre(\{\phi_1, \phi_2, \dots, \phi_m\})$  and postconditions  $Post(\{\psi_1, \psi_2, \dots, \psi_r\})$  in the Soar rule be well formed formulae. It also requires that a *valuation* function  $u(x)$  for the clocks  $x$  of the *TA* be defined over the non-negative reals. If the conditions specified in the requirements line are met, then the second line of the algorithm states that the property of the traces of the generated *TA* containing all the behaviors exhibited by the firing of the Soar rule is ensured.

---

**Algorithm 1.** Generate  $(S, s_0, \rightarrow)$  for  $TA = (L, l_0, C, A, E, I)$  from  $rname(V, Pre(\{\phi_1, \phi_2, \dots, \phi_m\}), Post(\{\psi_1, \psi_2, \dots, \psi_r\}))$

---

**Require:**  $\forall \phi(v_i), \psi(v_i) \in WFF, u: C \rightarrow \mathbb{R}_{\geq 0}$

**Ensure:**  $fire(rname) \subseteq traces(TA)$

```

 $l_0 \leftarrow \{Start\}$ 
 $L \leftarrow \{Start, Run\}$ 
 $s_0 \leftarrow (Start, u(x_0))$ 
 $S \leftarrow \{Start, Run\} \times u(x)$ 
 $I(Start) = \{True\}$ 
 $I(Run) = \{True\}$ 
 $G = \{ \}$ 
 $A = \{Run\_Rule\}$ 
for  $j = 1$  to  $m$  do
   $G \leftarrow G \wedge \phi_j$ 
end for
for  $k = 1$  to  $r$  do
   $A \leftarrow A \wedge \psi_j$ 
end for
if  $u \in I(Start)$  and  $L = \{Start\}$  then
   $e_1 = (Start, A, G, u \mapsto u', Run)$ 
end if
if  $u' \in I(Run)$  and  $L = \{Run\}$  then
   $e_2 = (Run, Run\_Rule, \{ \}, u' \mapsto u'', Start)$ 
end if
 $E = \{e_1, e_2\}$ 
 $S' \xleftarrow{u(x) \cup A} S$ 

return  $TA = (L, l_0, C, A, E, I)$ 

```

---

The next eight lines of the algorithm (lines 3–10) are used to initialize the elements of the  $TA$  and portions of its semantics, namely: (1) the set of initial locations  $l_0$ , (2) the set of locations  $L$ , (3) the set of initial states  $s_0$ , (4) the set of states  $S$ , (5) the invariant at the initial location, (6) the invariant at the location after the  $TA$  has executed its transition, (7) the set of actions of the  $TA$ , and (8) the set of guards for the  $TA$ . The Soar rule has two locations associated with it: (1)  $Start = val(V(u))$ , which describes the valuation of the state variables given in the Soar rule, before it has been fired, and (2)  $Run = val(V(u'))$ , which describes the valuation of the state variables given in rule after it has been fired. The state space is given by the cross production between the locations and clocks. There are no invariants at the locations, as currently non-determinism is handled in a sequential fashion, and the set of guards is initially set to empty. The set of actions for each  $TA$  created from a Soar rule has the base action  $Run\_Rule$ , which is generated by the global scheduler, and used to guarantee execution. The  $Run\_Rule$  action is received by all  $TA$ , and it forces all enabled  $TA$  (i.e.  $TAs$  whose guards are true, and thus, all Soar rules whose preconditions are true) to execute.

The first *for* loop (lines 11–13) is used to create the guard for the *TA* transition from the location *Start* to the location *Run*, and captures the *condition* portion of the Soar rule, by creating a conjunction of all of the preconditions (left-hand side) expressed by that rule. These are the preconditions that are needed for the Soar rule to fire, and are dependent on variables such as time. The second *for* loop (lines 14–16) is used to define the actions for the *TA* transition from the location *Start* to the location *Run*, and captures the *action* portion of the Soar rule, by creating a conjunction of all of the postconditions (right-hand side) expressed by that rule, along with the base action *Run.Rule*. These are the actions taken once the Soar rule fires (and the scheduler broadcast *Run.Rule* has been received), and act to change the state variables  $V$ , described in the Soar rule, of the system.

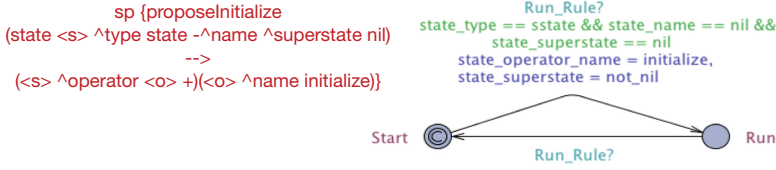
The next *if* statement (lines 17–19) creates the edge from the *Start* location to the *Run* location in the *TA* semantics. The origin and destination locations are *Start* and *Run* respectively. The set of actions  $A$  and guards  $G$  for this transition were defined by the previous two *for* loops. The clock is not reset on the edge, and thus its valuation advances from  $u$  to  $u'$ . Note that the conditional for forming the edge was that the initial clock valuation satisfied the invariant at the origin location of the edge. The following *if* statement (lines 20–22) creates the *TA* edge from the *Run* location to the *Start* location. This enables the *TA* to be reset after firing by the next broadcast of the *Run.Rule* action by the global scheduler. Thus, this mimics the behavior that the Soar rule is immediately able to be refired in the cognitive model. The origin and destination locations of the edge are *Run* and *Start* respectively. The *Run.Rule* action must be received for the reset to occur, and thus is the only action for the edge. There are no guards on this edge, and the clock is not reset, enabling time to progress from  $u'$  to  $u''$ . Lines 23–24 form the semantics for the *TA* by specifying the set of edges and transitions. Finally, in line 25, the algorithm returns the *TA* generated from the Soar rule  $rname(V, Pre(\{\phi_1, \phi_2, \dots, \phi_m\}), Post(\{\psi_1, \psi_2, \dots, \psi_r\}))$ .

We now walk through an example of this translation process as seen in Algorithm 1, for the rule given in Fig. 6.

## 5.2 Translation Implementation

The name of the Soar rule *proposeInitialize* (see Fig. 6) in Uppaal is a template name. To generate variable names, we linearize the working tree wherein each possible traversal ending in a useful value becomes the concatenated string of all visited identifiers during traversal (with an underscore as a delimiter). The preconditions in this Soar rule states that it: (1) does not have a name for the state  $s^{\wedge}name$  and (2) state  $s^{\wedge}superstate$  is *nil*. This is translated into the following guard:  $state\_name == nil$  and  $state\_superstate == nil$ . The action in this Soar rule is:  $state \langle s \rangle \hat{\text{operator}} \langle o \rangle \hat{\text{name}} \text{initialize}$ . This gets translated into the following TA action:  $s\_operator\_name = initialize$ .

The scheduler (Fig. 5) is designed to meet the following criteria: (1) It is configurable to meet different cognitive architectures, (2) Precondition satisfaction results in the selection of a rule to be executed, and (3) It tests the cognitive



**Fig. 6.** Mapping Soar to Uppaal

architecture goal condition to see if the goal has been reached. After the production rules have all been translated into timed automata, and the scheduler has been built, verification and validation activities can occur, as explained in the next section. Elements of non determinism due to learning (that changes rules) is challenging to translate into Uppaal as it is implicit in Soar.

## 6 Verification and Validation Efforts

### 6.1 Simulation Efforts for Validation of the Autonomous Pilot Agent

To test the Autonomous CoPilot Agent in a number of different scenarios, we connected the commercial X-plane aircraft simulation [31] with a shim that reads the relevant aircraft state variables (e.g., speed, altitude, attitude, position) and injects them into Soar’s working memory. The state variables are updated every 200 ms. The Soar agent captures the sensor value and then executes each rule within 100 ms. After experimenting with the simulation environment the change in velocity was set such that it changed every 150 ms with maximum acceleration. With this data sensor rate we minimize discrete jumps and attain a more or less continuous change in the value for the velocity. The rules for nominal takeoff and/or engine-out takeoff monitor the state of working memory to ensure that the appropriate actions are taken when conditions warrant. Figure 7 shows the connection between the aircraft state variables and the Java-based Soar Pilot Agent. This simulation was used to validate the autonomous copilot design for takeoff through multiple flown scenarios. Usability scenarios involved injecting engine faults at various times. Soar has an input/output link in its working memory which serves as the interface to different input output devices. The output branch for a Soar model has a memory element/node called speech. The autonomous agent adds children to this node; those children have literal string values. These values are sent to a text to speech engine as they appear thereby mimicking the interplay between the PF and PNF. The text to speech engine is a standalone Python application which is called by the Jsoar wrapper. These rules that initiate verbal interaction from the autonomous copilot are translated into Uppaal, with the verbal command translated as an update or action item for a variable representing the verbal communication.

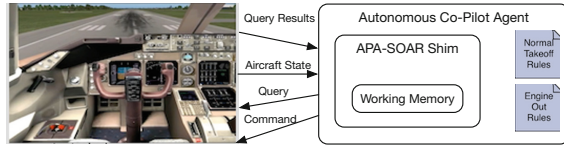


Fig. 7. Block diagram of the simulation configuration

## 6.2 Formal Verification of the Autonomous Pilot Agent

The Soar rules for the autonomous agent that modeled the procedures followed by a copilot were translated into Uppaal. The rules executed are based on inputs received from the flight simulator; this necessitated the creation of inputs such as airspeed. In order to provide changes to the airspeed, a new input template was created within the Uppaal model in which the airspeed was updated at every step of execution. We used a first order model for vehicle velocity which we plan to refine. This was followed by proving properties in Uppaal such as:

- Airspeed greater than 80 is followed by applying rule for airspeed alive  
**R1**  $\text{state\_io\_input\_link\_flightdata\_airspeed} == 80 \text{ -- } >$   
**state\\_operator\\_name == callaa**
- All paths eventually lead to calling out Airspeed Alive  
**R2**  $\text{A} \langle > \text{state\_operator\_name} == \text{callaa}$
- All paths eventually lead to calling out rotate  
**R3**  $\text{A} \langle > \text{state\_operator\_name} == \text{callrotate}$
- TakeOff shall be abandoned if there is engine out and velocity is below the threshold  
**R4**  $\text{state\_abandon} == \text{true} \text{ -- } > \text{applycallAbandon.0.Run}$

The properties were proven on two sets of models. The first model represented the nominal Takeoff without any failures. The second model represented the procedures followed for the engine-out use case. While verifying the above properties such as R1, R2 and R3 in Uppaal on the first model, we encountered an out of range exception, as shown in Fig. 8. This error was generated due to the fact that at each cycle of the execution, Soar looks for a change in state caused either by new working memory data being input, or modifications to working memory data caused by rule execution. When there is no change in state it is called an Impasse, and Soar attempts to generate a sub-goal to continue to make progress towards a solution. The creation and resolution of sub-goals requires a hierarchical decomposition of the problem space, which is not necessary for straightforward examples. Instead, we introduce a counting mechanism which forces a change in state. This is effectively a busy-wait state for new data in Soar. The output of this counting can be unbounded and was never captured in the Soar environment, as events always occurred in the environment before the variable would overflow. But this unbounded variable overflow was captured

while proving the property R1 in Uppaal on the nominal model. Hence, in periods where no events are taking place, it is possible for the copilot agent to *time out*, in some respects.

The properties verified on the off nominal model, such as R4, prove successfully indicating the copilot responds to the abandon event appropriately. Presently, the use case is deterministic so uncertainty was not included in the model and translation, but both Soar and Uppaal allows probabilistic models, thus there is the potential to include uncertainty in our future research efforts. This will enable evaluating the efficiency of reordering the set of rules when a contingency arises. This also enables us to model uncertainty due to sensor observation and modeling approximation along with reaction times taken by human agents.

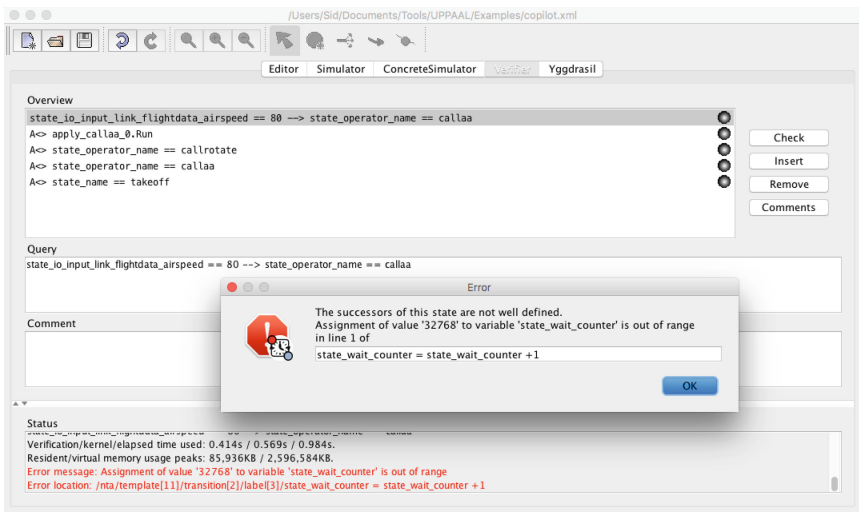


Fig. 8. Out of range error for translated Uppaal copilot agent

## 7 Conclusion and Future Work

Cognitive architectures have proven to be beneficial in the design of intelligent adaptive systems, as they allow the integration of learning algorithms as plug-ins within a defined architectural representation. Additionally, they allow representation of collaborative human-machine teaming by modeling the autonomous agents working with humans. Presently, systems designed with these cognitive architectures cannot be deployed for safety critical applications as the methods to assure correctness of their behavior are inadequate. A significant contribution of our work has been the development of an extensible framework for the design and formal verification of systems whose intelligent attributes can be modeled in

a cognitive architecture. We have formally described and implemented the automated model transformation that translates an intelligent agent into a formally verifiable temporal logic based model. This translation enables formal verification of cognitive models developed in cognitive architectures or rule based systems such as Soar. We plan to extend this research to fully integrate multiple learning methods that are capable of modifying rules within the system and formally verifying the resulting system. We also intend to model uncertainty in the system through the incorporation of probabilistic models. We plan to evaluate merging Soar rules into one Uppaal template as otherwise the number of Uppaal templates become fairly large.

## References


1. Newell, A., Shaw, J.C., Simon, H.A.: Report on a general problem-solving program. In: Proceedings of the International Conference on Information Processing, pp. 256–264 (1959)
2. Buchanan, B.G., Shortliffe, E.H.: Rule Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project. The Addison-Wesley Series in Artificial Intelligence. Addison-Wesley Longman Publishing Co., Inc., Boston (1984)
3. Anderson, J.R., Matessa, M., Lebiere, C.: ACT-R: a theory of higher level cognition and its relation to visual attention. *Hum.-Comput. Interact.* **12**(4), 439–462 (1997)
4. Laird, J.E.: The SOAR Cognitive Architecture. MIT Press, Cambridge (2012)
5. Sutton, R.L., Barto, B.: Reinforcement Learning. MIT Press, Cambridge (2008)
6. Mittal, S., Douglass, S.A.: Net-centric ACT-R based cognitive architecture with DEVS unified process. In: DEVS Symposium Spring Simulation Multiconference, Boston (2011)
7. Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: architecture-based self adaptation with reusable infrastructure. *Computer* **37**(10), 46–54 (2004)
8. Wen, M., Ehlers, R., Topcu, U.: Correct-by-synthesis reinforcement learning with temporal logic constraints. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (2015)
9. Sharifloo, A.M., Spoletini, P.: LOVER: Light-weight fOrmal Verification of adaptive systems at Run time. In: Păsăreanu, C.S., Salaün, G. (eds.) FACS 2012. LNCS, vol. 7684, pp. 170–187. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-35861-6\\_11](https://doi.org/10.1007/978-3-642-35861-6_11)
10. Curzon, P., Ruknas, R., Blandford, A.: An approach to formal verification of human computer interaction. *Form. Asp. Comput.* **19**, 513–550 (2007)
11. O’Conner, M., Tangirala, S., Kumar, R., Bhattacharyya, S., Sznaier, S., Holloway, L.: A bottom-up approach to verification of hybrid model-based hierarchical controllers with application to underwater vehicles. In: Proceedings of American Control Conference (2006)
12. Rocha, C., Cadavid, H., Muñoz, C., Siminiceanu, R.: A formal interactive verification environment for the plan execution interchange language. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 343–357. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-30729-4\\_24](https://doi.org/10.1007/978-3-642-30729-4_24)
13. Dowek, G., Munoz, C., Pasareanu, C.: A small-step semantics of PLEXIL (2008)

14. Dowek, G., Munoz, C., Pasareanu, C.: A formal analysis framework for PLEXIL. In: Proceedings of 3rd Workshop on Planning and Plan Execution for Real-World Systems (2007)
15. Dowek, G., Munoz, C., Rocha, C.: Rewriting logic semantics of a plan execution language. In: EPTCS, vol. 18, pp. 77–91 (2009)
16. Strauss, P.J.: Executable semantics for PLEXIL: simulating a task-scheduling language in Haskell. Masters thesis (2009)
17. Balasubramanian, D., Pasareanu, C., Whalen, M.W., Karsai, G., Lowry, M.R.: Polyglot: modeling and analysis for multiple statechart formalisms. In: Dwyer, M.B., Tip, F. (eds.) ISSTA. ACM (2011)
18. Verdejo, A., Martí-Oliet, N.: Two case studies of semantics execution in Maude: CCS and LOTOS. *Formal Methods Syst. Des.* **27**, 113–172 (2005)
19. Eskridge, T.C., Carvalho, M.M., Bhattacharyya, S., Vogl, T.: Verifiable autonomy final report. Technical report, Florida Institute of Technology and Rockwell Collins (2015)
20. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45657-0-29>
21. Uppaal website (2010). <http://www.uppaal.org>
22. Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M.: PVS: combining specification, proof checking, and model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 411–414. Springer, Heidelberg (1996). [https://doi.org/10.1007/3-540-61474-5\\_91](https://doi.org/10.1007/3-540-61474-5_91)
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](https://doi.org/10.1007/978-3-540-78800-3_24)
24. Larsen, K.G., Pettersson, P., Yi, W.: Model-checking for real-time systems. In: Reichel, H. (ed.) FCT 1995. LNCS, vol. 965, pp. 62–88. Springer, Heidelberg (1995). [https://doi.org/10.1007/3-540-60249-6\\_41](https://doi.org/10.1007/3-540-60249-6_41)
25. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL: a tool suite for automatic verification of real-time systems. *Theor. Comput. Sci.* (1996). RS-96-58
26. Alur, R., David, L.D.: A theory of timed automata. *Theor. Comput. Sci.* **126**, 183–235 (1999)
27. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: a model-checking tool for real-time systems. In: Ravn, A.P., Rischel, H. (eds.) FTRTFT 1998. LNCS, vol. 1486, pp. 298–302. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0055357>
28. Neogi, N.A.: Capturing safety requirements to enable effective task allocation between humans and automaton in increasingly autonomous systems. In: Proceedings of the AIAA Aviation Forum. 16th AIAA Aviation Technology, Integration, and Operations Conference (AIAA 2016-3594) (2016)
29. Code of Federal Regulations: Title 14 Aeronautics and Space. Federal Register, May 1962. <http://www.ecfr.gov/cgi-bin/text>
30. The Boeing Company: Boeing 737 pilots operating handbook. Continental Airlines, November 2002. <http://air.felisnox.com/view.php?name=737.pdf>
31. Official x-plane website (2016). <http://www.x-plane.com>





# Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C

Allan Blanchard<sup>1</sup>✉, Nikolai Kosmatov<sup>2</sup>, and Frédéric Loulergue<sup>3</sup> 

<sup>1</sup> Inria Lille — Nord Europe, Villeneuve d'Ascq, France  
allan.blanchard@inria.fr

<sup>2</sup> CEA, List, Software Reliability and Security Laboratory, PC 174,  
Gif-sur-Yvette, France  
nikolai.kosmatov@cea.fr

<sup>3</sup> School of Informatics Computing and Cyber Systems, Northern Arizona University,  
Flagstaff, USA  
frederic.loulergue@nau.edu

**Abstract.** Internet of Things (IoT) applications are becoming increasingly critical and require rigorous formal verification. In this paper we target Contiki, a widely used open-source OS for IoT, and present a verification case study of one of its most critical modules: that of linked lists. Its API and list representation differ from the classical linked list implementations, and are particularly challenging for deductive verification. The proposed verification technique relies on a parallel view of a list through a companion ghost array. This approach makes it possible to perform most proofs automatically using the Frama-C/WP tool, only a small number of auxiliary lemmas being proved interactively in the Coq proof assistant. We present an elegant segment-based reasoning over the companion array developed for the proof. Finally, we validate the proposed specification by proving a few functions manipulating lists.

**Keywords:** Linked lists · Deductive verification · Operating system  
Internet of Things · Frama-C

## 1 Introduction

Connected devices and services, also referred to as Internet of Things (IoT), are gaining wider and wider adoption in many security critical domains. This raises important security challenges, which can be addressed using formal verification.

This paper focuses on Contiki [10], a popular open-source operating system for IoT devices providing full low-power IPv6 connectivity, including 6TiSCH, 6LoWPAN, RPL, or CoAP standards. It is implemented in C with an emphasis on memory and power optimization, and contains a kernel linked to platform-specific drivers at compile-time. When Contiki was created in 2002, no particular attention was paid to security. Later, communication security was integrated, but formal verification of code was not performed until very recent case studies [19, 21].

```

1  struct list {
2      struct list *next;
3      int k; // a data field
4  };
5  typedef struct list ** list_t;
6  //Initialize a list
7  void list_init(list_t pLst);
8  //Get the length of a list
9  int list_length(list_t pLst);
10 //Get the first element of a list
11 void * list_head(list_t pLst);
12 //Get the last element of a list
13 void * list_tail(list_t pLst);
14 //Remove the first element of a list
15 void * list_pop (list_t pLst);
16 //Add an item to the start of a list.
17 void list_push(list_t pLst, void *item);
18 //Remove the last element of a list.
19 void * list_chop(list_t pLst);
20 //Add an item at the end of a list.
21 void list_add(list_t pLst, void *item);
22 //Duplicate a list (copy head pointer)
23 void list_copy(list_t dest, list_t src);
24 //Remove element item from a list
25 void list_remove(list_t pLst, void *item);
26 //Insert newitem after previtem in a list
27 void list_insert(list_t pLst,
28                 void *previtem, void *newitem);
29 //Get the element following item
30 void * list_item_next(void *item);

```

**Fig. 1.** API of the `list` module of Contiki (for lists with one integer data field)

The goal of this work is to perform deductive verification of the linked list module, one of the most critical modules of Contiki. It is performed using the ACSL specification language [4] and the deductive verification plugin WP of FRAMA-C [16]. Our approach is based on a parallel view of a linked list via a companion ghost array. Such a “flattened” view of the list avoids complex inductive predicates and allows for automatic proof of most properties, but requires to maintain a strong link between the companion array and the list it models. All proofs have been checked for several list structures. A few auxiliary lemmas have been proved in the Coq proof assistant [24], where a significant effort has been done to make their proofs robust and independent of the specific list structure. Finally, we have validated the proposed specification by proving a few functions manipulating lists.

**The contributions** of this work include

- formal specification of the `list` module<sup>1</sup> of Contiki in the ACSL specification language and its deductive verification using the WP plugin of FRAMA-C;
- a presentation of the underlying approach based on a companion ghost array;
- formal statement and proof of several lemmas useful for reasoning about this representation;
- pointing out an (unintended?) inconsistency in precondition of one function;
- a preliminary validation of the proposed specification of the module via a successful verification of a few annotated test functions dealing with lists.

**Outline.** The paper is organized as follows. Section 2 presents the specifics of the linked list module. Section 3 describes our verification approach and results. Section 4 provides some related work, while Sect. 5 gives the conclusion and future work.

## 2 The List Module of Contiki

Required by 32 modules and invoked more than 250 times in the core of the OS, the linked list module (`list`) is a crucial library in Contiki. It is used, for

<sup>1</sup> Complete annotated code available at <http://allan-blanchard.fr/code/contiki-list-verified.zip>.

instance, to implement the scheduler, where lists are used to manage timers and processes. Its verification is thus a key step for proving many other modules of the OS.

The API of the module is given in Fig. 1. Technically, it differs from many common linked list implementations in several regards. First, in Contiki an existing list (illustrated by the lower part of Fig. 3) is identified or modified through a dedicated list handler – a variable supposed to refer to the first list element – called `root` in Fig. 3. Copying one linked list into another is just copying such a list handler (cf. lines 22–23 in Fig. 1) without duplicating list elements. In a function call, an existing list is thus passed as a function parameter via a pointer referring to the handler (denoted in this paper by `pLst` and having a double pointer type `list_t`), rather than just the address of the first list element (i.e. a single pointer, contained in `root`) to make it possible to modify the handler in the function.

Second, being implemented in C (that does not offer templates), Contiki uses a generic mechanism to create a linked list for specific field datatypes using dedicated macros. The preprocessor transforms such a macro into a new list datatype definition. Lines 1–4 in Fig. 1 show the resulting definition for a list with one integer field. To be applicable for various types, the common list API treats list elements via either `void*` pointers or pointers to a trivial linked list structure (having only lines 1, 2, 4), and relies on (explicit and implicit) pointer casts. To make possible such a “blind” list manipulation using casts, the first field in any list element structure must be the pointer to the next list element (cf. line 2). That means that, for a user-defined type `struct some_list`, when a cell of this type is transmitted to the list API, the implementation first erases the type to `void*` and then casts it to `struct list*` to perform list manipulations and modifications. Note that according to the C standard, this implementation violates the strict-aliasing rule, since we modify a value of type `struct some_list` through a pointer to a type `struct list`. So the compilation of Contiki is configured to deactivate the assumption of strict-aliasing compliance.

Third, Contiki does not provide dynamic memory allocation, which is replaced by attributing (or releasing) a block in a pre-allocated array [19]. In particular, the size of a list is always bounded by the number of such blocks, and their manipulation does not invoke dynamic memory allocation functions.

Fourth, adding an element at the start or the end of a list is allowed even if this element is already in the list: in this case, it will first be removed from its previous position. Finally, the API is very rich: it can handle a list as a FIFO or a stack (lines 14–21), and supports arbitrary removal/insertion and enumeration (lines 24–30).

For all these reasons, the list module of Contiki appears to be a necessary but challenging target for verification with FRAMA-C/WP.

```

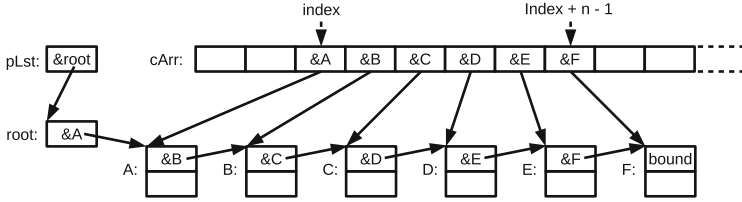
1 #define MAX_SIZE INT_MAX-1
2 /*@
3  requires \valid(pLst) ^ \valid(item) ^ \valid(cArr + (0 .. MAX_SIZE-1));
4  requires Linked: linked_n(*pLst, cArr, index, n, NULL);
5  requires 0 ≤ index ^ 0 ≤ n < MAX_SIZE ;
6  requires EnoughSpace: index + n + 1 ≤ MAX_SIZE;
7  requires Unique: ∀ ℤ y, z;
8     index ≤ y < index + n ^ index ≤ z < index + n ^ y ≠ z ⇒ cArr[y] ≠ cArr[z];
9  requires item_index == index_of(item, cArr, index, index+n) ;
10
11 assigns cArr[index .. index + n], cArr[item_index - 1]->next, item->next, *pLst;
12
13 behavior contains_item:
14   assumes ∃ ℤ i ; index ≤ i < index+n ^ cArr[i] == item ;
15   ensures linked_n(*pLst, cArr, index, n, NULL);
16   ensures unchanged{Pre,Post}(cArr, item_index + 1, index+n);
17   ensures array_swipe_right{Pre, Post}(cArr, index + 1, item_index + 1);
18   ensures cArr[item_index] == item;
19   ensures Unique: ...; // no repetitions in the list (see lines 7-8)
20
21 behavior does_not_contain_item:
22   assumes ∀ ℤ i ; index ≤ i < index+n ⇒ cArr[i] ≠ item ;
23   ensures linked_n(*pLst, cArr, index, n+1, NULL);
24   ensures array_swipe_right{Pre, Post}(cArr, index + 1, index + n + 1);
25   ensures cArr[item_index] == item;
26   ensures Unique: ...; // no repetitions in the list (similar to lines 7-8)
27
28 complete behaviors;
29 disjoint behaviors;
30 */
31 void list_push(list_t pLst, struct list *item,
32 /* ghost: */ struct list **cArr, int index, int n, int item_index)
33 {
34   struct list *l;
35   /*@ ghost int rem_n = item_index == index+n ? n : n-1 ;
36
37   list_remove(pLst, item, /* ghost: */ cArr, index, n, item_index);
38   /*@ assert array_swipe_left{Pre,Here}(cArr, item_index, index + rem_n);
39   /*@ assert unchanged{Pre,Here}(cArr, index, item_index);
40
41   /*@ ghost array_push(cArr, index, rem_n, list, *list, NULL);
42   /*@ assert array_swipe_right{Pre,Here}(cArr, index + 1, item_index + 1);
43   /*@ assert unchanged{Pre,Here}(cArr, item_index + 1, index + rem_n + 1);
44
45   /*@ assert linked(*pLst, cArr, index + 1, rem_n, NULL);
46
47   item->next = *pLst;
48   /*@ ghost cArr[item_index] = item ;
49
50   /*@ assert linked_n(item, cArr, index, rem_n+1, NULL);
51   *pLst = item;
52 }

```

Fig. 2. Function `list_push` adds given element `item` to the start of the list

### 3 The Verification Approach

This section presents our verification approach and results. Since the generic mechanism of type manipulation lies beyond the border of undefined behavior in C (cf. Sect. 2), formally speaking, a separate verification is necessary for any new list structure datatype as for a different API. That is why in this verification case study we use a precise definition of list structure. In the presentation of this paper, we assume that the list structure is defined by lines 1–4 of Fig. 1 and use



**Fig. 3.** Parallel view of a list prefix using a companion array formally defined by the `linked_n` predicate in Fig. 4

pointers to **struct** list in the verified functions instead of generic **void\*** pointers. To ensure that this choice of list structure is not a limitation of the approach, we additionally check that our specification and proof remain valid for other common list structures (where data fields are a pointer, or a structure containing three coordinates of a point). All proofs remain successful for the tested list structures.

### 3.1 Running Example

We will use the function `list_push` (see Fig. 2) to illustrate the specification and verification of the list module in the rest of this article. This function adds a given list element `item` at the beginning of the list whose list handler is referred to by `pLst`. As the API must ensure that each list element appears at most once in the list, this function first tries to remove `item` from the list (see line 37 of Fig. 2). After this operation, it is guaranteed that `item` is not in the list. Then the next field of `item` is set to point to the (previous) first element of the list, that is, `*pLst` (line 47). Finally, the handler `*pLst` is updated to point to the new first element `item` (line 51).

The annotated version of the function also takes some ghost parameters (given on line 32) supposed to be ignored during compilation and execution. Note that we include here ghost parameters as regular parameters “considered-as-ghost” by preceding them by a comment `/* ghost: */` rather than syntactically writing them as ghost parameters inside an annotation `/*@ ghost: <type><param>; */`. While being part of the ACSL specification language [4], ghost parameters are currently not yet supported by the public releases of FRAMA-C/WP. It means that in order to verify other modules of the OS using lists, we currently have to modify all list function calls to add such “considered-as-ghost” parameters. To obtain the executable code from the annotated code, only a slight modification of code is required for the moment: to remove these “considered-as-ghost” parameters clearly marked on a separate line (since all other annotations are already seen as comments by a compiler). The support of ghost parameters is in progress in the development version and should be available in the near future. With this support, it will only be necessary to add ghost parameters in the modules under verification using lists without modifying the C API.

```

1 /*@
2   inductive linked_n{L}(struct list *root, struct list **cArr,
3                       Z index, Z n, struct list *bound) {
4     case linked_n_bound{L}:
5       V struct list **cArr, *bound, Z index;
6       0 ≤ index ≤ MAX_SIZE ⇒ linked_n(bound, cArr, index, 0, bound);
7     case linked_n_cons{L}:
8       V struct list *root, **cArr, *bound, Z index, n;
9       0 < n ⇒ 0 ≤ index ⇒ 0 ≤ index + n ≤ MAX_SIZE ⇒
10        \valid(root) ⇒ root == cArr[index] ⇒
11        linked_n(root->next, cArr, index + 1, n - 1, bound) ⇒
12        linked_n(root, cArr, index, n, bound);
13   }
14 */

```

**Fig. 4.** Inductive predicate `linked_n` creating a link between the list prefix of size  $n$  of the linked list `root` and the segment of indices `index..index+n-1` in the companion array `cArr`, where the sublist boundary `bound` refers to the list element immediately following this prefix. If the list is of size  $n$ , `bound` is `NULL`.

### 3.2 List Representation by a Companion Ghost Array

Our verification approach relies on a parallel view of the list via a companion ghost array whose cells refer to the elements of the list (see Fig. 3). Basically, it allows us to transform most of the inductive properties required during the verification into simple universally quantified formulas. Such formulas are easier to handle by SMT solvers, and generally easier to write. The predicate `linked_n` (cf. Fig. 4) inductively builds the link between a list and a segment of the companion array that models it. We define a few auxiliary lemmas that help to deal with this property without having to reason by induction (this induction being hidden in the proofs of the lemmas, as we explain in Sect. 3.5).

More precisely, the inductive predicate `linked_n` establishes a relation between the prefix of length  $n$  of the list starting at `root` and the segment of size  $n$  in the companion array `cArr` starting from `index`. The relation is established at label `L`, which can be omitted being by default the current program point. The pointer `bound` refers to the list element immediately following the represented sublist, that is, to the  $(1+n)$ -th element in the list. We refer to it as the sublist *boundary*. For an empty list (cf. lines 4–6), `root` and `bound` must be equal, and the list is considered to be linked with the empty segment of the companion array starting at any `index`. If the segment size (and the list length)  $n$  is greater than 0 (cf. lines 7–12), the first element `root` must point to a valid list element and this element must be registered in `cArr` at `index`. Moreover, the remaining part of the list prefix (starting at `root->next`) must in turn be linked to the array segment of size  $n-1$  starting at `index+1` with the same boundary element `bound`. To verify that the list representation relation holds, we will have to update the ghost array each time we modify the linked list. It is done through ghost functions and instructions presented below.

Notice that the ACSL annotations in this paper are slightly pretty-printed:  $\mathbb{Z}$  denotes the type **integer** of mathematical integers in ACSL, while universal and existential quantifiers are replaced by  $\forall$  and  $\exists$ , respectively.

```

1 /*@
2 //Range down..up-1 in cArr and the contents of the corresponding list elements
3 //are the same at labels L1 and L2
4 predicate unchanged{L1,L2}{struct list **cArr,  $\mathbb{Z}$  down,  $\mathbb{Z}$  up} =
5    $\forall \mathbb{Z} i; \text{down} \leq i < \text{up} \Rightarrow$ 
6      $\backslash\text{at}(cArr[i], L1) == \backslash\text{at}(cArr[i], L2) \wedge$ 
7      $(\backslash\text{valid}\{L1\}(\backslash\text{at}(cArr[i], K)) \Rightarrow \backslash\text{valid}\{L2\}(\backslash\text{at}(cArr[i], L2))) \wedge$ 
8      $\backslash\text{at}(*cArr[i], L1) == \backslash\text{at}(*cArr[i], L2);$ 
9 */
10 /*@
11 //Range down..up-1 at label L2 is a right shift of down-1..up-2 at label L1
12 predicate array_swipe_right{L1, L2}{struct list **cArr,  $\mathbb{Z}$  down,  $\mathbb{Z}$  up} =
13    $\forall \mathbb{Z} i; \text{down} \leq i < \text{up} \Rightarrow$ 
14      $\backslash\text{at}(cArr[i], L2) == \backslash\text{at}(cArr[i-1], L1);$ 
15 */

```

**Fig. 5.** Predicates relating the contents of a list between two program points L1 and L2, expressed in terms of the companion array cArr

### 3.3 Formal Specification

To perform deductive verification of the list module in FRAMA-C/WP, we first provide a formal specification of the list API in the ACSL specification language [4]. Let us illustrate the formal specification for the function `list_push`, whose (simplified) contract is given in Fig. 2. The parameters on line 32 are considered to be ghost parameters.

The precondition is given by **requires** clauses on lines 3–9 in Fig. 2. First (cf. line 3), both the pointer `pLst` to the list handler and the list element `item` must be pointers to valid memory locations. The companion array is assumed to be valid for a large range of indices as well. The representation of list `pLst` by the companion array `cArr` is assumed on line 4. The segment of size `n` starting at `index` represents here the whole list (since the list boundary is equal to `NULL`). Lines 5–6 assume necessary domain constraints for `n` and `index`, in particular, the possibility to add one more element at the end of the segment in `cArr`. Lines 7–8 assume that any list element appears in the list at most once. Line 9 assumes that the ghost parameter `item_index` records the position of `item` in `cArr` (and thus in the list). It is computed by the logic function `index_of` whose definition is straightforward and not presented here. This function returns the position of `item` in `cArr` if it can be found in the segment of indices `index..(index+n-1)`, or one past the last segment `index+(n)` otherwise. Notice that the last two properties of the list are conveniently expressed in terms of the companion array.

The **assigns** clause on line 11 specifies the variables that the function is allowed to modify. For other postconditions, we distinguish two cases (defined in ACSL by behaviors, cf. lines 13 and 22): either `item` is already present in the list or not. An **assumes** clause defines the domain of application of each behavior. In both cases, the resulting list must be represented by the companion array. However, in the first case (behavior `contains_item`, lines 13–19), it must have the same size as before (cf. line 15), since we first remove the element and then add it again at the start of the list, whereas in the second case (behavior `does_not_contain_item`, lines 21–26), the resulting list will grow (cf. line

23). In both cases, `item` is added at the start of the list (cf. lines 18, 25). To express the conditions on other elements, we need two additional predicates given in Fig. 5.

The predicate `unchanged` in Fig. 5 states that, between two programs points at labels `L1` and `L2`, in the range `down..up-1` both the elements of the companion array and the contents of the corresponding list elements are the same, and the validity of the list elements has been preserved between `L1` and `L2`. The predicate `array_swipe_right` states that the cells in the range `down-1..up-2` at label `L1` are shifted to the right to become the range `down..up-1` at label `L2`.

Thanks to these predicates, we can specify the remaining parts of the contract. In the behavior `does_not_contain_item`, where `item` is originally not in the list, `item` is simply placed at the beginning of the list. In terms of the companion array, it is expressed as a shift of the corresponding segment to the right (cf. line 24). In the behavior `contains_item`, `item` is removed at its previous position `item_index` and added at the start of the list. In terms of the companion array, the segment of indices `item_index+1..index+n-1` remains unchanged between the **Pre** state (before the call) and the **Post** state (after the call, cf. line 16). However, the segment of indices `index..item_index-1` at **Pre** is shifted to the right to the range of indices `index+1..item_index` at **Post** (cf. line 17). This precisely describes the desired postconditions for the list.

The postconditions on lines 19 and 26 state that the uniqueness of list elements (lines 7–8) is preserved by the function in both cases for the new companion array segment representing the list (in the second case, the new segment is one element longer).

Lines 28–29 indicate that the given behaviors are complete and disjoint: they cover all possible cases and cannot apply at the same time. Thanks to these annotations, the completeness and disjointness of behaviors are checked by FRAMA-C/WP.

A quite important part of the contract that we have removed in this simplified version is separation conditions. The ACSL language and the FRAMA-C/WP tool support such separation properties. Each element of the list must be spatially separated from any other to guarantee that the list is well formed. In ACSL, this property can be expressed as follows:

```
1  $\forall \mathbb{Z} y, z; \text{index} \leq y < \text{index} + n \wedge \text{index} \leq z < \text{index} + n \wedge y \neq z \Rightarrow$ 
2   \separated( *(array + y), *(array + z) );
```

Furthermore, the list elements must be separated from the companion ghost array to guarantee that any operation on the companion array does not impact the linked list itself and *vice versa*. It can be expressed as follows:

```
1  $\forall \mathbb{Z} y; \text{index} \leq y < \text{index} + n \Rightarrow$ 
2   \separated( *(array + y), array + (0 .. MAX_SIZE - 1) );
```



```

1 /*@
2  requires \valid(pLst) ^ \valid(cArr + (0 .. MAX_SIZE - 1));
3  requires EnoughSpace: index + n + 1 ≤ MAX_SIZE;
4  requires Linked: linked_n(root, cArr, index, n, bound);
5  requires Unique: ...; // no repetitions in the list
6
7  assigns cArr[index .. index + n];
8
9  ensures Unique: ...; // no repetitions in the list
10 ensures SwipeR: array_swipe_right(Pre, Post)(cArr, index + 1, index + n + 1);
11 ensures Linked: linked_n(root, cArr, index + 1, n, bound);
12 */
13 void array_push(struct list **cArr, int index, int n,
14                list_t pLst, struct list *root, struct list *bound)
15 {
16     int i = index + n; // next index to assign in cArr
17     struct list *le = bound; // current boundary between left and right segments
18     /*@
19     loop invariant IInBounds: index ≤ i ≤ index + n ;
20     loop invariant UnchangedLeft: unchanged(Pre, Here)(cArr, index, i);
21     loop invariant ISwipeR: array_swipe_right(Pre, Here)(cArr, i + 1, index+n+1);
22     loop invariant LeftLinked: linked_n(root, cArr, index, i - index, le);
23     loop invariant RightLinked: linked_n(le, cArr, i + 1, n - (i - index), bound);
24     loop assigns i, le, cArr[index+1 .. index + n];
25     loop variant i - index;
26     */
27     while (i > index){
28         cArr[i] = cArr[i-1]; // shift next cArr element
29         i--;
30
31         // compute new boundary
32         if (i == index ) le = root;
33         else le = cArr[i];
34     }
35 }

```

Fig. 6. Ghost function `array_push` shifts ghost array `cArr` to the right

These properties<sup>2</sup> are systematically included as preconditions and postconditions in each function contract of the API. In the precondition of `list_push`, we also need to state that list elements and companion array cells are separated from the list handler referred to by `pLst` and the element `item` to be added. The complete version of the contracts is available online.

### 3.4 Ghost Functions

To maintain the list representation by the companion array, we have to update the array each time the linked list is modified. It is done through ghost functions and instructions. Let us illustrate it for the `list_push` function. First, `item` is removed if needed (line 37 in Fig. 2). Line 35 defines the length `rem_n` of the resulting list (and companion array segment) after this operation: `n` if `item` was not present, and `n-1` otherwise. Lines 38-39 provide two assertions on the two segments around the deleted element: the left segment up to the initial position `item_index` of `item` is unchanged (line 39), while the right segment – non empty only if `item` was present in the list – is obtained as a left shift of a

<sup>2</sup> Some of which will be no longer necessary when ghost parameters will be fully supported in FRAMA-C.

segment at state **Pre**. (The definition of a left shift is similar to the right shift and is omitted here.) These properties come from the contract of `list_remove`.

Next, `item` has to be added at the beginning of the list. We make the choice to keep the same starting position `index` for the segments that model the list in the precondition and in the postcondition (cf. lines 4, 15 and 23). To create some place for a new first element in the beginning of the segment, we use the ghost function `array_push` (cf. line 41 in Fig. 2).

The `array_push` function (see Fig. 6) shifts the segment of the companion array one cell to the right. Starting from the end of the segment (at position  $i = \text{index} + n$  in the array), it moves the element at position  $i - 1$  to position  $i$  (cf. lines 27–34). The boundary between the shifted and not-yet-shifted segments is maintained in variable `le`. The function contact specifies the shift (cf. line 10) and preserves the link of the list with the shifted segment (cf. lines 4, 11). Line 7 indicates the segment of the companion array that is modified. Uniqueness properties (lines 5, 9) and separation properties (not presented in the simplified version given in Fig. 6) should also be included in the contract (as explained in Sect. 3.3). The loop contract (lines 18–26) is necessary to reason about segment manipulation as we will explain in Sect. 3.5.

After the call of `array_push` in the ghost code on line 41 in Fig. 2, the cell at position `index` in the companion array is not used. Combining the partial left shift due to a list element removal by the call to `list_remove` and the complete right shift by the call to `array_push`, we obtain the properties of the assertions on lines 42–43 in Fig. 2. The interested reader will easily check them by considering separately the case of each behavior. Moreover, we have the representation property on line 45 for the shifted segment.

Thanks to these properties, after connecting the list element `item` to the original list (line 47) and recording `item` in the companion array at position `index` in a ghost assignment (line 48), we obtained the representation of the list started with `item` by the segment of the companion array from position `index` with `rem_n + 1` elements and with a boundary `NULL`. After the assignment of the list handler `*pLst` on line 51, the required representation of the resulting list by the companion array is reconstructed (cf. lines 15, 23).

These examples illustrate a benefit of the companion array for this specification – and more precisely for expressing predicates like `unchanged`, `array_swipe_left` and `array_swipe_right`, and separation properties. All these properties can be directly expressed using the companion array. It means that we do not need induction any more to reason about them.

### 3.5 Auxiliary Lemmas and Proofs

Inductive properties are generally not well handled by SMT solvers since most of the time, a proof that involves inductive properties requires reasoning by induction. On the contrary, SMT solvers are efficient when they just have to instantiate lemmas that directly state implications between known properties. Thus, providing lemmas about inductive properties can significantly improve the treatment of inductive properties in a proof by requiring reasoning by induction

```

1 /*@
2 lemma stay_linked{L1, L2}:
3   ∀ struct list *root, **cArr, *bound, ℤ i, n ;
4     linked_n{L1} (root, cArr, i, n, bound) ⇒
5     unchanged{L1, L2}(cArr, i, i+n) ⇒
6     linked_n{L2} (root, cArr, i, n, bound);
7 */
8 /*@
9 lemma linked_split_segment:
10  ∀ struct list *root, **cArr, *bound, *b0, ℤ i, n, k;
11    n > 0 ⇒ k ≥ 0 ⇒
12    b0 == cArr[i + n - 1]->next ⇒
13    linked_n(root, cArr, i, n + k, bound) ⇒
14    (linked_n(root, cArr, i, n, b0) ∧ linked_n(b0, cArr, i + n, k, bound));
15 */
16 /*@
17 lemma linked_merge_segment:
18  ∀ struct list *root, **cArr, *bound, *b0, ℤ i, n, k;
19    n ≥ 0 ⇒ k ≥ 0 ⇒
20    (linked_n(root, cArr, i, n, b0) ∧ linked_n(b0, cArr, i + n, k, bound)) ⇒
21    linked_n(root, cArr, i, n + k, bound);
22 */

```

**Fig. 7.** Examples of lemmas about the `linked_n` predicate

only in the proofs of the lemmas. We already successfully experimented a similar approach to count values within a range of indices in an array [6].

For example, a very simple property (illustrated by Fig. 7, lines 2–6) currently not handled by SMT solvers in our verification is the fact that if a list `root` is linked to a companion array `cArr` at a given program point `L1` (line 4), and if the list representation (i.e. the corresponding companion array segment and pointed list elements) has not changed between `L1` and another program point `L2` (line 5), then the list `root` is still linked to `cArr` at program point `L2` (line 6). For example, when we modify some element in the array, this lemma is useful to ensure that a `linked_n` relation still holds for other, unmodified segments before and after this element.

Two more subtle properties are the facts that we can *split* the `linked_n` property at a given valid index into two segments, or conversely *merge* two consecutive segments into a longer one. The `linked_split_segment` lemma (cf. Fig. 7) states that if a list starting from `root` is linked to the segment in `cArr` at index `i` with size `n+k` and reaches a given boundary `bound`, we can split it into two relations: the first one linking `root` to the segment at position `i` with `n` elements and reaching boundary `b0`, and the second linking `b0` to the segment at position `i+n` with `k` elements and reaching boundary `bound`, where the intermediary boundary `b0` is defined by line 12. Conversely, if we have two consecutive linked segments in the same companion array, where the boundary of the first segment refers to the first list element of the second list, we can deduce the `linked_n` relation for a longer segment that combines them (cf. lemma `linked_merge_segment` in Fig. 7).

This kind of property is, for example, useful for the verification of the `array_push` function. Indeed, in the loop, when the assignment `cArr[i] = cArr[i-1]` at line 28 (of Fig. 6) overwrites the cell at position

$i$ , we have to maintain the representation of the list prefix before this position (line 22 of the invariant), and the *split* lemma allows this because we can detach the end using it. At the same time, this assignment puts a new element just before the beginning of the segment specified by the invariant of line 23, that gives a new `linked_n` predicate, progressively built by the function. As we know that this list element is linked to the first cell of this range, the lemma *merge* allows to combine them into a longer segment.

In the current formalization, these lemmas have been proved using the Coq proof assistant [5,24]. Most of the lemmas related to the predicate `linked_n` have been proved by induction on the predicate itself or on the size  $n$  of the list. The induction principle used is an induction principle similar to the basic induction principle on Peano natural numbers but applied on the positive subset of relative numbers. The axiomatic function `index_of` is defined on a lower and an upper bound. It was therefore necessary to reason by induction on their difference. One challenge for these proof scripts was to make them very robust so they can be valid for various versions of the list structure. In particular the unchanged predicate does not take the same number of arguments for different versions of the list structure.

In addition to these lemmas, four assertions are not proved directly by the SMT solvers. Two are proved in Coq: they are basically applications of lemmas, under some conditions. Two are proved by the recently introduced Interactive Proof Editor (TIP) of the WP plugin. It offers a new panel that focuses on one goal generated by WP, and allows the user to visualize the goal to be proved. The user can then interactively decompose a complex proof into smaller pieces by applying tactics, and the pieces are proved by SMT solvers. In our case two assertions were proved inside each branch of a conditional but were not automatically proved just after the conditional. The proof using TIP was straightforward.

### 3.6 Results of the Verification

In this work, we have annotated and verified all functions of the list module, except `list_insert` (as detailed below). In total, for about 176 lines of C code in the module (excluding macros), we wrote 46 lines for ghost functions, and about 1400 lines of annotations, including about 500 lines for contracts and 240 lines for logic definitions and lemmas. We did not specifically try to minimize the number of intermediate assertions: they were added to explicitly state the expected local properties and to help the automatic proof, and some of them could probably be removed. For this annotated version of the module, the verification using FRAMA-C/WP generates 798 goals. This number includes 108 goals for the verification of absence of runtime errors that are often responsible for security vulnerabilities and have also been carefully checked by FRAMA-C/WP. It also includes 24 auxiliary lemmas (that is, in total only about 3.3% of properties). The 24 lemmas are proved using Coq v.8.6.1. Out of the 774 remaining goals, almost all are automatically discharged by SMT solvers, except for 4 goals that are proved interactively (as mentioned in Sect. 3.5). In this work, we used FRAMA-C v.16 Sulfur and the solvers Alt-Ergo v.1.30 (with direct

translation from WP and via Why3), as well as Z3 v.4.5 and CVC3 v.2.4.1 (via Why3).

The verification helped to identify an inconsistency for the remaining function, `list_insert` (cf. lines 26–28 in Fig. 1), with respect to the assumptions of other functions (and, in a sense, to itself). This function adds an element `new_item` into a list just after a given element `prev_item`. If the element `prev_item` is NULL, the function directly calls `list_push`, meaning that if the element is already there, it is removed and then added to the start. However, if `prev_item` is present in the list, the function directly adds `new_item` after `prev_item` without removing a previous instance of `new_item` from the list (if any). It shows that the uniqueness property is in general not preserved by the function, but in some cases it is. Thus this function does not respect a contract consistent with the other functions. We decided to ask the authors of Contiki to clarify this potentially dangerous behavior, that could for example allow to break the integrity of the list<sup>3</sup>. Moreover, in the entire code of Contiki, we have found only one call to `list_insert`, and not a single one in the core part of the system.

Unit tests could have permitted to identify such a bug. However, one difficulty with tests (that is also the cause of many security bugs) is the fact that we tend to test valid scenarios rather than invalid ones.

### 3.7 Validation of Specification

To get confidence in the proposed specification, we have implemented 15 simple valid test functions<sup>4</sup> manipulating lists, and tried to prove simple properties on lists in them using the proposed contracts of the list module functions. The results show that in all tests, the correct properties were successfully proved by WP.

We have also implemented 15 invalid tests. Each invalid function is a variant of a valid one where we have altered the contract (including such dangerous cases as violations of separation, or validity, or uniqueness), the ghost values or the code itself. For those functions, the verification leads, as expected, to proof failure.

This gives us further confidence that the proposed contracts can be successfully used for a larger deductive verification of Contiki. This step has also allowed us to detect and fix some minor deficiencies in the contracts at the latest phases of the work and can be recommended to be systematically performed for all similar verification projects.

## 4 Related Work

To our knowledge, there is no other specification of a linked list API using ghost arrays. This approach has two main advantages. First such specifications

<sup>3</sup> The issue can be found at: <https://github.com/contiki-ng/contiki-ng/issues/254>.

<sup>4</sup> Included in the online archive with the annotated code.

may be more readable to developers not used to formal specifications written in a more functional style using inductive type definitions. Secondly, it makes these specifications close to be usable in a context of dynamic verification [8], in particular using the E-ACSL plugin of FRAMA-C [17]. One drawback is that the support of ghost function parameters is not yet available, and another more important one, is that the specifications should contain assertions stating the separation of the actual memory cells and the ghost array.

Dross and Moy [9] present the proof of an implementation of red-black trees in SPARK that involves underlying arrays. They also rely on ghost code for the proof, the main difference in our work is that the array we use for representation is only part of the specification and thus the cells can be allocated independently of this array using another policy, while they use arrays as the actual implementation of the trees. On the proof side, the main difference is that we rely on Coq to prove simple lemmas that can be used automatically by SMT solvers, while they use the so-called auto-active proof [18] to avoid the use of an interactive proof assistant.

In this context, separation logic [23] is more suitable than the Hoare logic in which the WP plugin of FRAMA-C is based. Tools based on such a kind of logic may therefore be more suitable for the verification of a linked list API, for example VeriFast [15] or the Verified Software Toolchain [1, 3]. The former has been used in several industrial case studies [22] while the latter has been mainly used to verify cryptography related software [2, 26] but also a message passing system [20]. We are not aware of efforts dedicated to the verification of linked list functions, but the example gallery of VeriFast<sup>5</sup> and the VST case studies do include linked list function specifications and verification. They are based on a logic list data structure and an inductive predicate relating the memory and such a logical data structure, as initially done by Reynolds [23]. Reynolds reasoned on sequences of instructions rather than functions. He thus mostly expressed loop invariants. This has an impact on the style of the specifications. In this case, an existential quantification of logical data structures is convenient. VeriFast has the concept of patterns, that are kind of free variables in preconditions, that are bound to values during symbolic execution, and that can be used in postconditions. Reynolds' style of specification is therefore possible in VeriFast. In the case of ACSL and JML, when specifying functions or methods, an existential quantification in the precondition only binds a variable in the precondition: the scope does not extend to the postcondition. Using Reynolds' style of specification is therefore not possible. In the case of JML, to specify Java methods on a linked list data structure, Gladisch and Tyszberowicz [12] used a pure observer method that takes a list object and an index, and returns the object at that index in the list. The methods they consider are simpler than the list API of Contiki, but essentially our ghost arrays can be seen as observations of the linked lists.

VeriFast and VST are based on *concurrent* separation logic [7, 14]. It is difficult to compare the specifications because most VeriFast case studies take into account concurrency. For some of the examples related to linked lists, being

<sup>5</sup> <https://people.cs.kuleuven.be/~bart.jacobs/verifast/examples/>.

based on separation logic does not seem to have a significant impact on the size of specifications with respect to the specifications we have.

For the verification, FRAMA-C is the most automated. Moreover, in addition to the automated and interactive provers it natively supports, it can output verification conditions to many different provers using Why3ML [11] as an intermediate verification language. This eases the verification, and makes it more trustworthy. VeriFast is based on its embedded SMT solver Redux and can also use Z3. VST is a framework for Coq. The specifications can be written in the Gallina language of Coq, making them very expressive. However, even with the dedicated tactics, proofs are less automated in VST than in FRAMA-C or VeriFast.

One strong point of VST is that its logic is fully formalized in Coq and its soundness has been proved in Coq. It makes it the safer framework. Both for the WP Plugin and VeriFast, there exist some results about the correctness of subsets of the tools [13, 25].

## 5 Conclusion and Future Work

The expansion of devices connected to the Internet raises many security risks. One promising way to tackle this challenge is to use formal methods, which is one of the goals of the EU H2020 VESSEDIA project. This paper reports on verification of the list module of Contiki, which is one of the most critical modules of the operating system. It requires, in the context of C language, to deal with linked data structures and memory separation that are still hard to handle automatically.

Our verification approach relies on a companion ghost array and some ghost code. Although the idea of using ghost code is not new, it appears to be highly beneficial in this context, allowing for an elegant reasoning over the companion array and for an automatic proof of the great majority of goals: more than 96% of goals in this case study have been proved automatically. Imposing a limit on the size of the companion array (and therefore, the lists it models) is not a limitation since list sizes in Contiki are always bounded. The verification of the list module, intensively used by other modules of Contiki, opens the way to formal verification of higher-level modules depending on it. All proofs of this case study have been checked for several list structures. Moreover, we have implemented several test functions working with lists, and validated the proposed specification by obtaining a successful proof for correct properties, and a proof failure in the erroneous cases.

In future work, we plan to start the verification of higher-level modules of Contiki. This verification is planned to be done using both the deductive verification tool FRAMA-C/WP and the value analysis tool FRAMA-C/EVA. The latter can handle linked data-structures slightly better than the former, but again, it is more suitable to reason about arrays. While it would have been impossible to prove the equivalence between our array representation and the lists using EVA (that is not really meant to treat functional properties), it can still benefit of

this verification since we can now directly reason about arrays. Note however that it still requires to instrument user code, and to verify that lists are only modified through the API.

**Acknowledgment.** This work was partially supported by a grant from CPER DATA and the project VESSEDIA, which has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 731453. The authors thank the FRAMA-C team for providing the tools and support, as well as Patrick Baudin, François Bobot and Loïc Correnson for many fruitful discussions and advice. Many thanks to the anonymous referees for their helpful comments.

## References

1. Appel, A.W.: Verified software toolchain. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 1–17. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-19718-5\\_1](https://doi.org/10.1007/978-3-642-19718-5_1)
2. Appel, A.W.: Verification of a cryptographic primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* **37**(2), 7:1–7:31 (2015)
3. Appel, A.W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S., Leroy, X.: *Program Logics for Certified Compilers*. Cambridge University Press, Cambridge (2014)
4. Baudin, P., Cuq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. <http://frama-c.com/acsl.html>
5. Bertot, Y., Castéran, P.: *Interactive Theorem Proving and Program Development*. Springer, Heidelberg (2004). <https://doi.org/10.1007/978-3-662-07964-5>
6. Blanchard, A., Kosmatov, N., Lemerre, M., Loulergue, F.: A case study on formal verification of the anaxagoras hypervisor paging system with Frama-C. In: Núñez, M., Güdemann, M. (eds.) FMICS 2015. LNCS, vol. 9128, pp. 15–30. Springer, Cham (2015). <https://doi.org/10.1007/978-3-319-19458-2>
7. Brookes, S., O’Hearn, P.W.: Concurrent separation logic. *ACM SIGLOG News* **3**(3), 47–65 (2016)
8. Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes* **31**(3), 25–37 (2006)
9. Dross, C., Moy, Y.: Auto-active proof of red-black trees in SPARK. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 68–83. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-57288-8\\_5](https://doi.org/10.1007/978-3-319-57288-8_5)
10. Dunkels, A., Gronvall, B., Voigt, T.: Contiki – a lightweight and flexible operating system for tiny networked sensors. In: LCN 2004. IEEE (2004)
11. 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](https://doi.org/10.1007/978-3-642-37036-6_8)
12. Gladisch, C., Tyszbrowicz, S.: Specifying linked data structures in JML for combining formal verification and testing. *Sci. Comput. Program.* **107–108**, 19–40 (2015)
13. Herms, P., Marché, C., Monate, B.: A certified multi-prover verification condition generator. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 2–17. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-27705-4\\_2](https://doi.org/10.1007/978-3-642-27705-4_2)



14. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 353–367. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-78739-6\\_27](https://doi.org/10.1007/978-3-540-78739-6_27)
15. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-20398-5\\_4](https://doi.org/10.1007/978-3-642-20398-5_4)
16. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. *Form. Asp. Comput.* **27**(3), 573–609 (2015). <http://frama-c.com>
17. Kosmatov, N., Signoles, J.: A lesson on runtime assertion checking with Frama-C. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 386–399. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-40787-1\\_29](https://doi.org/10.1007/978-3-642-40787-1_29)
18. Leino, K.R.M., Moskal, M.: Usable auto-active verification (2010). <http://fm.csl.sri.com/UV10/>
19. Mangano, F., Duquennoy, S., Kosmatov, N.: Formal verification of a memory allocation module of Contiki with FRAMA-C: a case study. In: Cuppens, F., Cuppens, N., Lanet, J.-L., Legay, A. (eds.) CRiSIS 2016. LNCS, vol. 10158, pp. 114–120. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-54876-0\\_9](https://doi.org/10.1007/978-3-319-54876-0_9)
20. Mansky, W., Appel, A.W., Nogin, A.: A verified messaging system. *Proc. ACM Program. Lang.* **1**(OOPSLA), 87:1–87:28 (2017)
21. Peyrard, A., Duquennoy, S., Kosmatov, N., Raza, S.: Towards formal verification of Contiki: analysis of the AES-CCM\* modules with Frama-C. In: RED-IoT 2018, Co-located with EWSN 2018. ACM (2018, to appear)
22. Philippaerts, P., Mühlberg, J.T., Penninckx, W., Smans, J., Jacobs, B., Piessens, F.: Software verification with VeriFast: industrial case studies. *Sci. Comput. Program.* **82**, 77–97 (2014)
23. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS), pp. 55–74. IEEE Computer Society (2002)
24. The Coq Development Team: The Coq proof assistant. <http://coq.inria.fr>
25. Vogels, F., Jacobs, B., Piessens, F.: Featherweight VeriFast. *Log. Methods Comput. Sci.* **11**(3), 1–57 (2015)
26. Ye, K.Q., Green, M., Sanguansin, N., Beringer, L., Petcher, A., Appel, A.W.: Verified correctness and security of mbedTLS HMAC-DRBG. In: ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 2007–2020. ACM, New York (2017)



# An Executable Formal Framework for Safety-Critical Human Multitasking

Giovanna Broccia<sup>1</sup>(✉) , Paolo Milazzo<sup>1</sup>, and Peter Csaba Ölveczky<sup>2</sup>

<sup>1</sup> Department of Computer Science, University of Pisa, Pisa, Italy  
{giovanna.broccia,milazzo}@di.unipi.it

<sup>2</sup> University of Oslo, Oslo, Norway

**Abstract.** When a person is concurrently interacting with different systems, the amount of cognitive resources required (cognitive load) could be too high and might prevent some tasks from being completed. When such human multitasking involves safety-critical tasks, for example in an airplane, a spacecraft, or a car, failure to devote sufficient attention to the different tasks could have serious consequences. To study this problem, we define an executable formal model of human attention and multitasking in Real-Time Maude. It includes a description of the human working memory and the cognitive processes involved in the interaction with a device. Our framework enables us to analyze human multitasking through simulation, reachability analysis, and LTL and timed CTL model checking, and we show how a number of prototypical multitasking problems can be analyzed in Real-Time Maude. We illustrate our modeling and analysis framework by studying the interaction with a GPS navigation system while driving, and apply model checking to show that in some cases the cognitive load of the navigation system could cause the driver to keep the focus away from driving for too long.

## 1 Introduction

These days we often interact with multiple devices or computer systems at the same time. Such *human multitasking* requires us to repeatedly shift attention from task to task. If some tasks are safety-critical, then failure to perform the tasks correctly and timely—for example due to cognitive overload or giving too much attention to other tasks—could have catastrophic consequences.

A typical scenario of safety-critical human multitasking is when a person interacts with a safety-critical device/system while using other less critical devices. For example, pilots have to reprogram the flight management system while handling radio communications and monitoring flight instruments [11]. Operators of critical medical devices, such as infusion pumps, often have to retrieve patient-specific parameters by accessing the hospital database on a different device while configuring the safety-critical device. Finally, a driver often interacts with the GPS navigation system and/or the infotainment system while driving.

Human multitasking could lead to cognitive overload (too much information to process/remember), resulting in forgetting/mistaking critical tasks. For

example, [16] reports that during a routine surgery, the ventilator helping the patient to breathe was turned off to quickly take an X-ray without blurring the picture. However, the X-ray jammed, the anesthesiologist went to fix the X-ray but forgot to turn on the ventilator, leading to the patient's death. In another example, [8] analyzes the cause of 139 deaths when using an infusion pump, and finds that operator distraction caused 67 deaths, whereas problems with the device itself only caused 10 deaths. Similar figures and examples can be found in the context of aviation [2] and car driving [10].

In addition to cognitive overload, human multitasking could also lead to ignoring the critical tasks for too long while focusing attention on less critical tasks. For instance, while reprogramming the flight management system, the pilot could miss something important on the flight instruments. If the interface of the virtual clinical folder requires the user's attention for too long, it can cause the operator to make some mistake in the infusion pump setup. An infotainment system that attracts the driver's attention for too long could cause a car accident.

There is therefore a clear need to analyze not only the functionality of single devices (or networks of devices), but also to analyze whether a human can safely use multiple devices/systems at the same time. Such study requires understanding how the human cognitive processes work when interacting with multiple systems and how human attention is directed at the different tasks at hand. In particular, the main cognitive resource to be shared among concurrent tasks is the human *working memory*, which is responsible for storing and processing pieces of information necessary to perform all the concurrent tasks.

In this paper we propose a formal executable model of human multitasking in safety-critical contexts. The model is specified in Real-Time Maude [18]. It is a significant modification and extension of the cognitive framework proposed by Cerone for the analysis of interactive systems [7]. As in that work, our model includes the description of the human working memory and of the other cognitive processes involved in the interaction with a device. The main difference is that Cerone only considered the interaction with a *single* device, whereas we focus on analyzing human multitasking. In contrast to [7], our framework also captures the limitations of a human's working memory (to enabling reasoning about hazards caused by cognitive overload) and includes timing features (to analyze, e.g., whether a critical task is ignored for too long).

After providing some background on human attention and multitasking and Real-Time Maude in Sect. 2, we present our Real-Time Maude model of safety-critical human multitasking in Sect. 3. Section 4 explains how Real-Time Maude can be used to analyze prototypical properties in human multitasking. We illustrate our formal modeling and analysis framework in Sect. 5 by studying the use of a GPS navigator while driving. We apply model checking to show that in some cases: (i) the cognitive load of the navigator interface could cause the driver to keep the focus away from driving for too long, and (ii) the working memory sharing between concurrent tasks can lead to overloading situations causing failures in one of the tasks. Finally, Sect. 6 discusses related work, and Sect. 7 gives some concluding remarks.

## 2 Preliminaries

**Human Selective Attention and Multitasking.** The *short-term memory* is the component in human memory that is most involved in interactions with computers, and is then called the *working memory* (WM). It is a cognitive system with a limited capacity responsible for the transient holding, processing, and manipulation of information. Different hypotheses about the WM all agree that it can store a limited amount of items, and that it is responsible for both processing and storage activities. The amount of information—which can be digits, words, or other meaningful items—that the WM can hold is  $7 \pm 2$  items [17].

Maintaining items in the WM requires human attention. Memory items are remembered longer if they are periodically refreshed by focusing on them. Even when performing a single task, in order not to forget something stored in the WM, the task has to be interleaved with memory refreshment. The most successful psychological theory in terms of explaining experimental data is the Time-Based Resource Sharing Model [3]. It introduces the notion of *cognitive load* (CL) as the temporal density of attentional demands of the task being performed. The higher the CL of a task, the more it distracts from refreshing memory. According to [3], when the frequency of basic activities in a task is constant, the CL of the task equals  $\sum a_i n_i / T$ , where  $n_i$  is the number of task basic activities of type  $i$ ,  $a_i$  represents the difficulty of such activities, and  $T$  is the duration of the task.

Several studies show that the attentional mechanisms involved in WM refreshment are also the basis of multitasking. In particular, [12] describes the roles of the WM, the CL, and attention when executing a “main” task concurrently with a “distractor” task. It is shown that when the CL of the distractor task increases, the interaction with the main task could be impeded.

In [5] we use the cognitive load and two other factors, the task’s *criticality level* and *waiting time* (the time the task has been ignored by the user), to define a measure of task attractiveness called the *task rank*. The higher the task rank, the more likely the user will focus on it. Modeling attention switching based on parameters like CL, criticality level, and waiting time agrees with current understanding of human attention. In [5] we use this task rank to define an algorithm for simulating human attention. We studied the case of two concurrent tasks, and found that the task more likely to complete first is the one with the highest cognitive load, which is consistent with relevant literature (e.g., [3, 12]).

**Real-Time Maude.** Real-Time Maude [18] extends Maude [9] to support the executable formal specification and analysis of real-time systems in rewriting logic. Real-Time Maude provides a range of formal analysis methods, including simulation, reachability analysis, and LTL and timed CTL model checking.

A Real-Time Maude module specifies a *real-time rewrite theory* [19]  $(\Sigma, E \cup A, IR, TR)$ , where:

- $\Sigma$  is an algebraic *signature*; that is, declarations of *sorts*, *subsorts*, and *function symbols*, including a data type for time, which can be discrete or dense.

- $(\Sigma, E \cup A)$  is a *membership equational logic theory*, with  $E$  a set of (possibly conditional) equations, written  $\text{eq } t = t'$  and  $\text{ceq } t = t' \text{ if } \text{cond}$ , and  $A$  a set of equational axioms such as associativity, commutativity, and identity.  $(\Sigma, E \cup A)$  specifies the system's state space as an algebraic data type.
- $IR$  is a set of *labeled conditional rewrite rules* specifying the system's local transitions, each of which has the form  $\text{cr1 } [l] : t \Rightarrow t' \text{ if } \bigwedge_{j=1}^m u_j = v_j$ , where  $l$  is a *label*. Such a rule specifies an *instantaneous transition* from an instance of  $t$  to the corresponding instance of  $t'$ , provided the condition holds.
- $TR$  is a set of *tick rewrite rules*  $\text{cr1 } [l] : \{t\} \Rightarrow \{t'\} \text{ in time } \tau \text{ if } \text{cond}$ , which specify that going from the *entire* state  $t$  to state  $t'$  takes  $\tau$  time units.

The mathematical variables in equations and rewrite rules are either declared using the keyword `vars`, or are introduced on-the-fly and have the form  $\text{var} : \text{sort}$ . We refer to [9] for more details on the syntax of Real-Time Maude.

A declaration `class C | att1 : s1, ..., attn : sn` declares a class  $C$  with attributes  $\text{att}_1$  to  $\text{att}_n$  of sorts  $s_1$  to  $s_n$ . An *object* of class  $C$  is represented as a term  $\langle O : C \mid \text{att}_1 : \text{val}_1, \dots, \text{att}_n : \text{val}_n \rangle$  of sort `Object`, with  $O$  the object's *identifier*, and  $\text{val}_1$  to  $\text{val}_n$  the current values of the attributes  $\text{att}_1$  to  $\text{att}_n$ . The state of an object-oriented specification is a term of sort `Configuration`, and is a *multiset* of objects and messages. For example, the rewrite rule

```
cr1 [1] : < 01 : C | a1 : x1, a2 : 02, a3 : z, a4 : y1 >
         < 02 : C | a1 : x2, a2 : 01, a3 : w, a4 : y2 >
=>
         < 01 : C | a1 : x1 + w + z, a2 : 02, a3 : z, a4 : y1 >
         < 02 : C | a1 : x2 + z, a2 : 01, a3 : w, a4 : y2 > if z <= w
```

defines a family of transitions involving two objects 01 and 02 of class  $C$ , and updates the attribute `a1` of both objects. Attributes whose values do not change and do not affect the next state of other attributes or messages, such as `a4`, need not be mentioned in a rule. Attributes that are unchanged, such as `a2` and `a3`, can be omitted from right-hand sides of rules.

*Formal Analysis.* Real-Time Maude's *timed rewrite* command simulates *one* of the many possible system behaviors from the initial state by rewriting the initial state up to a certain duration. The *search* command

```
(utsearch [[n]] t =>* pattern [such that cond].)
```

uses a breadth-first strategy to search for (at most  $n$ ) states that are reachable from the initial state  $t$ , match the *search pattern*, and satisfy *cond*. If the arrow  $\Rightarrow!$  is used instead of  $\Rightarrow^*$ , then Real-Time Maude searches for reachable *final* states, that is, states that cannot be further rewritten.

A command `(find latest t =>* pattern [such that cond] with no time limit.)` explores all behaviors from the initial state  $t$  and finds the longest time needed to reach the desired state (for the *first* time in a behavior).

Real-Time Maude is also equipped with unbounded and time-bounded *linear temporal logic model checker* which analyzes whether *each* behavior (possible up to some duration) satisfies a linear temporal logic formula, and with a *timed CTL* model checker [15] to analyze *timed* temporal logic properties.

### 3 A Formal Model of Human Multitasking

This section presents our Real-Time Maude model of human multitasking. Due to space restrictions, we only show parts of the specification, and refer to our longer report [4] and the full executable specification available at <http://www.di.unipi.it/msvbio/software/HumanMultitasking.html> for more detail.

We model human multitasking in an object-oriented style. The state consists of a number of **Interface** objects, representing the interfaces of the devices/systems with which a user interacts, and an object of class **WorkingMemory** representing the user’s working memory. Each interface object contains a **Task** object defining the task that the user wants to perform on that interface.

#### 3.1 Classes

*Interfaces.* We model an interface as a transition system. Since we follow a user-centric approach, the state of the interface/system is given by what the human *perceives* it to be. For example, I may perceive that an ATM is ready to accept my debit card by seeing a welcoming message on the ATM display. A perception/state may not last forever: after entering my card in the slot, I will only perceive that the ATM is waiting for my PIN code for 8 min, after which the ATM will display a “Transaction cancelled” message. The term  $p$  for time  $t$  denotes that the user will perceive  $p$  for time  $t$ , after which the perception becomes `expired(p)`.

A transition of an interface has the form  $p_1 \text{ -- } action \text{ -->} p_2$ . If I perceive that the machine is ready to receive my card, I can perform an action `enterCard`, and the ATM will then display that I should type my PIN code: `ATMready -- enterCard --> typePIN for time 480`. Interface transitions are represented as a ;-separated set of single interface transitions. An interface is represented as an object instance of the following class:

```
class Interface | task : Object,          transitions : InterfaceTransitions,
                previousAction : DefAction, currentState : InterfaceState .
```

where the attribute `transitions` denotes the transitions of the interface; `task` denotes the task object (see below) representing the task that the user wants to perform with the interface; `previousAction` is the previous action performed on the interface (useful for analysis purposes); and `currentState` is (the user’s perception of) the state of the device. (See [4] for the data types involved.)

*Tasks.* Instead of seeing a task as a sequence of basic tasks that cannot be further decomposed, we find it more natural to consider a task to be a sequence of subtasks, where each subtask is a sequence of basic tasks. For example, the task of withdrawing money at an ATM may consist of the following sequence of subtasks: insert card; type PIN code; type amount; retrieve card; and, finally, retrieve cash. Some of these subtasks consist of a sequence of basic tasks: the subtask “type PIN code” consists of typing 4 digits and then “OK,” and so does the subtask “type amount.” We therefore model a

task as a ‘:.’-separated sequence of subtasks, where each subtask is modeled as a sequence of basic tasks of the form  $inf_1 \mid p_1 ==> action \mid inf_2 \text{ duration } \tau \text{ difficulty } d \text{ delay } \delta$ , where  $inf_1$  is some knowledge,  $p_1$  is a perception (state) of the interface,  $\tau$  is the time needed to execute the task, and  $d$  is the *difficulty* of the basic task. If my working memory contains  $inf_1$  and I perceive  $p_1$ , then I can perform the interface transition labeled *action*, and as a result my working memory forgets  $inf_1$  and stores  $inf_2$ . A basic task may not be enabled immediately: you cannot type your PIN code immediately after inserting your card. The (minimum) time needed before the basic task can be executed is given by the delay  $\delta$ , which could also be the time needed to switch from one task to another. A basic task could be

```
needCash | ATMready ==> enterCard | cardInMachine
  duration 3 difficulty 1/8 delay 0.
```

That is, after performing the action `enterCard` you “forget” that you need cash, and instead store in working memory that the card is in the machine.

As mentioned in Sect. 2, the next task that is given a person’s attention is a function of: the cognitive loads of the current subtasks<sup>1</sup>, the *criticality level* of each task (a person tends to focus more frequently on safety-critical tasks than on other tasks), and the time that an enabled task has *waited* to be executed. For example, driving a car has a higher criticality level than finding out where to go, which has higher criticality level than finding a good radio station. To compute the “rank” of each task, a task object should contain these values, and is therefore represented as an object instance of the following class `Task`:

```
class Task | subtasks : Task,      waitTime : Time,    status : TaskStatus,
  cognitiveLoad : Rat,    criticalityLevel : PosRat .
```

The `subtasks` attribute denotes the *remaining* sequence of subtasks to be performed; `waitTime` denotes how long the next basic task has been enabled; `cognitiveLoad` is the cognitive load of the subtask currently executing; and `criticalityLevel` is the task’s criticality level. For analysis purposes, we also add an attribute `status` denoting the “status” of the task, which is either `notStarted`, `ongoing`, `abandoned`, or `completed`.

*Working Memory.* The working memory is used when interacting with the interfaces, and can only store a limited number of information items. We model the working memory as an object of the following class:

```
class WorkingMemory | memory : Memory,    capacity : NzNat .
```

---

<sup>1</sup> Since we now consider *structured* tasks and add delays to basic tasks, we redefine the cognitive load of a task to be  $\sum \frac{d_i t_i}{t_i + dly_i}$ , where  $d_i$ ,  $t_i$  and  $dly_i$  denote the difficulty, duration and delay of each basic task  $i$  of the *current subtask*. The cognitive load of a task therefore changes every time a new subtask begins, and remains the same throughout the execution of the subtask.

where `capacity` denotes the maximal number of elements that can be stored in memory at any time. The attribute `memory` stores the content of the working memory as a map  $I_1 \mapsto mem_1 ; \dots ; I_n \mapsto mem_n$  of sort `Memory`, assigning to each interface  $I_j$  the *set*  $mem_j$  of items in the memory associated to interface  $I_j$ . An element in  $mem_j$  is either a *cognition* (see [4] for an explanation), a basic piece of *information*, such as `cardInMachine`, or a desired *goal* `goal(action)`. The goal defines the goal of the interaction with the interface, which is to end up performing some final action, such as `takeCash`.

### 3.2 Dynamic Behavior

We formalize human multitasking with rewrite rules that specify how attention is directed at the different tasks, and how this affects the working memory. In short, whenever a basic task is enabled, attention is directed toward the task/interface with the highest *task rank*, and a basic task/action is performed on that interface. The rank of a non-empty task is given the function `rank` defined as follows<sup>2</sup>:

```

eq rank(< I : Interface | task :
  < TASK : Task | subtasks : ((INF1 | P1 ==> DACT | INF2 duration
    NZT difficulty PR delay T2) BTL)
    :: OTHER-SUB-TASKS,
    waitTime : T, cognitiveLoad : CL,
    criticalityLevel : PR2 > >,
  (I |-> goal(ACT) INF-SET) ; MEMORY)
= if T2 == 0 then PR2 * CL * (T + 1) else 0 fi.

```

A task which is not yet enabled (the remaining delay `T2` of the first basic task is greater than 0) has rank 0. The `rank` function refines the task rank function in [5], and should therefore be consistent with results in psychology.

The following tick rewrite rule models the user performing a basic task (if it does not cause memory overload, and the action performed is not the goal action) with the interface with the highest rank of all interfaces (`bestRank(...)`):

```

cr1 [interacting] :
{OTHER-INTERFACES
  < I : Interface | task :
    < TASK : Task | subtasks : ((INF1 | P1 ==> DACT | INF2 duration NZT
      difficulty PR delay 0) BASIC-TASKS)
      :: OTHER-SUB-TASKS,
      waitTime : T1,          cognitiveLoad : CL,
      criticalityLevel : PR2, status : TS >,
      transitions : (P1 -- DACT --> (P2 for time TI2)) ; TRANSES,
      currentState : (P1 for time TI), previousAction : DACT2 >
    < WM : WorkingMemory | memory : MEMORY ; (I |-> INF1 goal(ACT) INF-SET),
      capacity : CAP >}

```

<sup>2</sup> We do not show the variable declarations, but follow the convention that variables are written in all capital letters.



```

=>
{idle(OTHER-INTERFACES, NZT)
  < I : Interface | task :
    < TASK : Task | subtasks : (if BASIC-TASKS /= nil
      then (BASIC-TASKS :: OTHER-SUB-TASKS)
      else OTHER-SUB-TASKS fi),
      waitTime : 0,
      status : (if TS == notStarted then ongoing else TS fi),
      cognitiveLoad : (if BASIC-TASKS /= nil then CL else
        cogLoad(first(OTHER-SUB-TASKS)) fi) >,
      currentState : (P2 for time TI2), previousAction : DACT >
  < WM : WorkingMemory | memory : MEMORY ; (I |-> INF2 goal(ACT) INF-SET) >}
in time NZT
  if assess(DACT2,P1) /= danger /\ (DACT /= ACT)
    /\ card(MEMORY ; (I |-> INF2 goal(ACT) INF-SET)) <= CAP
    /\ rank(< I : Interface | >,
      (MEMORY ; (I |-> INF1 goal(ACT) INF-SET)))
      == bestRank(< I : Interface | > OTHER-INTERFACES,
        (MEMORY ; (I |-> INF1 goal(ACT) INF-SET))).

```

The user perceives that the state of interface  $I$  is  $P1$ . The next basic task can be performed if information  $INF1$  is associated with this interface in the user's working memory, and the interface is (perceived to be) in state  $P1$ . The user then performs the basic task labeled  $DACT$ , which leads to a new item  $INF2$  stored in working memory, while  $INF1$  is forgotten. This rule is only enabled if the remaining delay of the basic task is 0 and the user has a goal associated with this interface. If the basic task performed is the last basic task in the subtask, we set the value of `cognitiveLoad` to be the cognitive load of the next subtask.

The first conjuncts in the condition say that the rule can only be applied when the user does not assess a danger in the current situation and when the action performed is not the goal action. Since  $INF1$  and/or  $INF2$  could be the empty element `noInfo`, the rule may increase the number of items stored in working memory (when  $INF1$  is `noInfo`, but  $INF2$  is not). The third conjunct in the condition ensures that the resulting knowledge does not exceed the capacity of the working memory. The last conjunct ensures that the current interface should be given attention: it has the highest rank among all the interfaces.

The duration of this tick rule is the duration  $NZT$  of the executing basic task. During that time, every other task idles: the "perception timer" and the remaining delay of the first basic task are decreased according to elapsed time, and the waiting time is increased if the basic task is enabled (see [4] for details).

If performing the basic task would exceed the capacity of the memory, some other item in the memory is nondeterministically forgotten, so that items associated to the current interface are only forgotten if there are no items associated to other interfaces. (This is because maintaining information in working memory requires the user's attention, and user attention is on the current task, so it is more natural that items of the other tasks are forgotten first.) The following rule shows the case when an item for a different interface is erased from memory.

Since a mapping is associative and commutative, *any* memory item INF3 associated with *any* interface I2 different from I could be forgotten. This rule is very similar to the rule above, and we only show the differences:

```

cr1 [interactingForgetSomethingOtherInterface] :
  { ... < I : Interface | task : < TASK : Task | ... > ... >
    < WM : WorkingMemory | memory : (I |-> INF1 goal(ACT) INF-SET) ;
                                   (I2 |-> INF3 INF-SET2) ; MEMORY,
                                   capacity : CAP >}
=>
  { ... < I : Interface | task : < TASK : Task | ... > ... >
    < WM : WorkingMemory | memory : (I |-> INF2 goal(ACT) INF-SET) ;
                                   (I2 |-> INF-SET2) ; MEMORY >}

  in time NZT
  if ... /\ card((I |-> INF2 goal(ACT) INF-SET)
                ; (I2 |-> INF3 INF-SET2) ; MEMORY) > CAP /\ ...

```

A similar rule removes an arbitrary item from the memory associated with the current interface if the memory does not store any item for another interface.

If each “next” basic task has a remaining delay, then time advances until the earliest time when the delay of some basic task reaches 0:

```

cr1 [tickAllIdling] :
  {ALL-INTERFACES
   < WM : WorkingMemory | memory: MEMORY ; (I |-> goal(ACT) INF-SET) >}
=>
  {idle(ALL-INTERFACES, MIN-DELAY)
   < WM : WorkingMemory | >} in time MIN-DELAY
  if MIN-DELAY := minDelay(ALL-INTERFACES).

```

where MIN-DELAY is a variable of a sort NzTime of non-zero time values.

The following rule concerns only the interface: sometimes the interface state comes with a timer (e.g., the ATM only waits for a PIN code for eight minutes). When this timer expires, an instantaneous rule changes the interface state (e.g., display “Ready” when the machine has waited too long for the PIN):

```

r1 [timeout] :
  {REST
   < I : Interface | transitions : (expired(P1) -- DACT --> IS) ; TRANSES,
                                   currentState : expired(P1) >}
=>
  {REST < I : Interface | currentState : IS, previousAction : DACT >}.

```

Our report [4] explains the rewrite rules when the goal action is performed (the **status** becomes **completed**), when the user changes her cognition (“mind”), and when the user perceives danger (the **status** becomes **abandoned**).

## 4 Analyzing Safety-Critical Human Multitasking

This section explains how Real-Time Maude can be used to analyze whether a human is able to perform a given set of tasks successfully. In particular, we focus on the following potential problems that could happen when multitasking:

1. A critical task may be ignored for too long because attention is given to other tasks. For example, it is not good if a driver does not give attention to driving for 15s because (s)he is focusing on the infotainment system.
2. A task, or a crucial action in a task, is not completed on time, since too much attention has been given to other tasks. For example, a pilot should finish all pre-flight tasks before taking off, and a driver should have entered the destination in the GPS before the first major intersection is reached.
3. Other tasks' concurrent use of working memory may cause the user to forget/misremember memory items that are crucial to complete a given task.

The initial state should have the form

```
{initializeCognLoad(
  < um : WorkingMemory | memory : interface1 |-> goal(action1) otherItems1 ; ... ;
                                interfacen |-> goal(actionn) otherItemsn,
                                capacity : capacity >
  < interface1 : Interface | task :
    < task1 : Task | subtasks : (b111 ... b11l1) :: ... :: (b1m11 ... b1m1j1),
                                waitTime : 0, cognitiveLoad : 0, criticalityLevel : cl1,
                                status : notStarted >
    transitions : trans1, previousAction : noAction, currentState : perc1 >
  ...
  < interfacen : Interface | task :
    < taskn : Task | subtasks : ..., waitTime : 0, cognitiveLoad : 0,
                                criticalityLevel : cln, status : notStarted >
    transitions : transn, previousAction : noAction, currentState : percn >}}
```

where:  $interface_k$  is the name of the  $k$ -th interface;  $task_k$  is the task to be performed with/on  $interface_k$ ;  $b_{k_i_j}$  is the  $j$ -th basic task of the  $i$ -th subtask of  $task_k$ ;  $cl_k$  is the criticality level of  $task_k$ ;  $trans_k$  are the transitions of  $interface_k$ ;  $action_k$  is the goal action to be achieved with  $interface_k$ ;  $otherItems_k$  are other items initially in the memory for  $interface_k$ ;  $perc_k$  is the initial perception (“state”) of  $interface_k$ ; and  $capacity$  is the number of items that can be stored in working memory. The function `initializeCognLoad` initializes the `cognitiveLoad` attributes by computing the cognitive load of the first subtask of each task.

The first key property to analyze is: Is it possible that an (enabled) task  $t$  is ignored continuously for at least time  $\Delta$ ? This property can be analyzed in Real-Time Maude as follows, by checking whether it is possible to reach a “bad” state where the `waitTime` attribute of task  $t$  is at least  $\Delta$ :<sup>3</sup>

<sup>3</sup> The variable `A:AttributeSet` captures the other attributes in *inner* objects.

```
(utsearch [1] initialState =>*
  {REST:Configuration < I:InterfaceId : Interface | task :
    < t : Task | waitTime : T:Time, A:AttributeSet > >}
  such that T:Time >= Δ.
```

where the variable `REST:Configuration` matches the other objects in the state.

The second key property is checking whether a certain task  $t$  is guaranteed to finish before time  $T$ . This can be analyzed using Real-Time Maude’s `find latest` command, by finding the longest time needed to reach status `completed`:

```
(find latest initialState =>*
  {REST:Configuration < I:InterfaceId : Interface | task :
    < t : Task | status : completed, A:AttributeSet > >}
  with no time limit.)
```

We can also use the `find latest` command to find out the longest time needed for a task  $t$  to complete the specific action  $act$ :

```
(find latest initialState =>*
  {REST:Configuration < I:InterfaceId : Interface | previousAction : act >}
  with no time limit.)
```

We can analyze whether it is guaranteed that a task  $t$  will be completed by searching for a “bad” *final* state where the status of the task is not `completed`:

```
(utsearch [1] initialState =>!
  {REST:Configuration < I:InterfaceId : Interface | task :
    < t : Task | status : TS:TaskStatus, A:AttributeSet > >}
  such that TS:TaskStatus /= completed.)
```

If we want to analyze whether it is guaranteed that *all* tasks can be completed, we just replace  $t$  in this command with a variable `I2:TaskId`.

If a safety-critical task cannot be completed, or completed in time, we can check whether this is due to the task itself, or the presence of concurrent “distractor” tasks, by analyzing an initial state *without* the distractor tasks.

## 5 Example: Interacting with a GPS Device While Driving

This section illustrates the use of our modeling and analysis framework with an example of a person who interacts with a GPS navigation device while driving.

We have two interfaces: the car and the navigation system. The task of driving consists of the three subtasks (i) start driving, (ii) drive to destination, and (iii) park and leave the car. The first subtask consists of the basic tasks of inserting the car key, turning on the ignition, and start driving; subtask (ii) describes a short trip during which the driver wants to perform a basic driving action at most every three time units; and subtask (iii) consists of stopping the car and removing the key when we have arrived at the destination. The driving task can be formalized by the following `Task` object:

```

< driving : Task | subtasks :
((noInfo | carOff ==> insertKey | keyInserted duration 1 difficulty 3/10 delay 0)
 (noInfo | carOn ==> turnKey | noInfo duration 1 difficulty 2/10 delay 0)
 (noInfo | carReady ==> startDrive | noInfo duration 1 difficulty 2/10 delay 2)) ::
((noInfo | straightRoad ==> straight | noInfo duration 1 difficulty 1/10 delay 3)
 (noInfo | straightRoad2 ==> straight | noInfo duration 1 difficulty 1/10 delay 3)
 (noInfo | curveLeft ==> turnLeft | noInfo duration 1 difficulty 4/10 delay 3)
 (noInfo | curveRight ==> turnRight | noInfo duration 1 difficulty 2/10 delay 3)
 (noInfo | straightRoad3 ==> straight | noInfo duration 1 difficulty 1/10 delay 3)
 (noInfo | straightRoad4 ==> straight | noInfo duration 1 difficulty 1/10 delay 3))
::
((noInfo | destination ==> stopCar | noInfo duration 2 difficulty 2/10 delay 2)
 (keyInserted | carStopped ==> pickKey | noInfo duration 2 difficulty 1/10
                    delay 0)),
waitTime : 0, status : notStarted, criticalityLevel : 6/10, cognitiveLoad : 0 >

```

The interface of the car is formalized by the following Interface object:

```

< car : Interface | transitions :
(carOff -- insertKey --> carOn) ; (carReady -- startDrive --> straightRoad) ;
(carOn -- turnKey --> carReady) ; (straightRoad -- straight --> straightRoad2) ;
(straightRoad2 -- straight --> curveLeft) ; (curveLeft -- turnLeft --> curveRight) ;
(curveRight -- turnRight --> straightRoad3) ;
(straightRoad3 -- straight --> straightRoad4) ;
(straightRoad4 -- straight --> destination) ; (destination -- stopCar --> carStopped) ;
(carStopped -- pickKey --> carOff) ; (carReady -- noAction --> carOff),
task : ... , previousAction : noAction, currentState : carOff >

```

For the GPS navigator, we assume that to enter the destination the user has to type at least partially the address. The navigator then suggests a list of possible destinations, among which the user has to select the right one. Therefore, the GPS task consists of three subtasks: (i) start and choose city; (ii) type the initial  $k$  letters of the desired destination; and (iii) choose the right destination among the options given by the GPS.

If the user types the entire address of the destination, the navigator returns a short list of possible matches; if (s)he types fewer characters, the navigator returns a longer list, making it harder for the user to find the right destination. We consider two alternatives: (1) the driver types 13 characters and then searches for the destination in a short list; and (2) the driver types just four characters and then searches for the destination in a longer list. The GPS task for case (1) is modeled by the following Task object:

```

< findDestination : Task | subtasks :
((noInfo | gpsReady ==> typeSearchMode | noInfo duration 1 difficulty 1/10
                    delay 0))
::
((noInfo | chooseCity ==> selectCity | noInfo duration 2 difficulty 5/10 delay 2))
::
((noInfo | typing1 ==> typeSomething | noInfo duration 1 difficulty 3/10 delay 3)
 (noInfo | typing2 ==> typeSomething | noInfo duration 1 difficulty 3/10 delay 0)

```

```

...
(noInfo | typing13 ==> pushSearchBtn | noInfo duration 1 difficulty 3/10 delay 0))
::
((noInfo | searching ==> chooseAddress | noInfo duration 2 difficulty 2/10
                                delay 0)),
waitTime : 0, status : notStarted, criticalityLevel : 3/10, cognitiveLoad : 0 >

```

Case (2) is modeled similarly, but with only four typing actions before pushing the search button. In that case, the last basic task (choosing destination from a larger list) has duration 5 and difficulty  $\frac{6}{10}$ .

The GPS interface in case (1) is defined by the following `Interface` object:

```

< gps : Interface | transitions :
  (gpsReady -- typeSearchMode --> chooseCity); (chooseCity -- selectCity --> typing1);
  (typing1 -- typeSomething --> typing2); (typing2 -- typeSomething --> typing3);
  ...
  (typing13 -- pushSearchBtn --> searching); (searching -- chooseAddress --> gpsReady),
task : ... , previousAction : noAction, currentState : gpsReady >

```

The initial state of the working memory is

```

< wm : WorkingMemory | capacity: 5, memory: (car |-> goal(pickKey)) ;
                                (gps |-> goal(chooseAddress)) >

```

We use the techniques in Sect. 4 to analyze our models, and first analyze whether an enabled driving task can be ignored for more than six seconds:

```

Maude> (utsearch [1] {initState} =>* {< car : Interface | task :
    < driving : Task | waitTime : T:Time, A:AttributeSet > >
    REST:Configuration} such that T:Time > 6.)

```

Real-Time Maude finds no such bad state when the driver types 13 characters. However, when the driver only types four characters, the command returns a bad state: the driver types the last two characters and finds the destination in the long list without turning her attention to driving in-between.

Sometimes even a brief distraction can be dangerous. For example, when the road turns, a delay of three time units in making the turn could be dangerous. We check the longest time needed for the driver to complete the `turnLeft` action:

```

Maude> (find latest {initState} =>*
    {REST:Configuration < car : Interface | previousAction: turnLeft >}
    with no time limit.)

```

Real-Time Maude shows that the left turn is completed at time 21. However, the same analysis with an initial state *without* the GPS interface object and task shows that an undistracted driver finishes the left turn at time 17.

Finally, to analyze potential memory overload, we modify the GPS task so that the driver must remember the portion of address already written: a new item is added to the working memory after every three characters typed.

We then check whether all tasks are guaranteed to be completed in this setting, by searching for a *final* state in which some task is not completed:

```
Maude> (utsearch [1] {initState2} =>! {< I:InterfaceId : Interface | task :
  < T:TaskId : Task | status : TS:TaskStatus, A:AttributeSet > >
  REST:Configuration} such that TS:TaskStatus /= completed.)
```

This command finds such an undesired state: `keyInserted` could be forgotten when the driver must remember typing; in that case, the goal action `pickKey` is not performed, and we leave the key in the car. The same command with our “standard” model of GPS interaction does not find any final state with an uncompleted task pending.

## 6 Related Work

There has been some work on applying “computational models” to study human attention and multitasking. The ACT-R architecture, an executable rule-based framework for modeling cognitive processes, has been applied to study, e.g., the effects of distraction by phone dialing while driving [20] and the sources of errors in aviation [6]. Recent versions of ACT-R handle human attention in accordance with the theory of concurrent multitasking proposed in [21]. The theory describes concurrent tasks that can interleave and compete for resources. Cognition balances task execution by favoring least recently processed tasks.

Other computational models for human multitasking include the *salience, expectancy, effort and value (SEEV)* model [23] and the *strategic task overload management (STOM)* model [22, 24]. Both have been validated against data collected by performing experiments with real users using simulators. Although dealing with human multitasking, the SEEV and STOM models are specifically designed to describe (sequential) visual scanning of an instrument panel, where each instrument may serve different tasks. The multitasking paradigms underlying SEEV and STOM are different from the one we consider in this paper, which is not *sequential scanning* but *voluntary task switching* [1].

The above systems (and other similar approaches) have all been developed in the context of cognitive psychology and neuroscience research. They do not provide what computer scientists would call a formal model, but are typically based on some mathematical formulas and an implementation (in Lisp in the case of ACT-R) that supports only simulation. In contrast, we provide a formal model that can be not only simulated, but also subjected to a range of formal analyses, including reachability analysis and timed temporal logic model checking.

On the formal methods side, Gelman et al. [13] model a pilot and the flight management system (FMS) of a civil aircraft and use WMC simulation and SAL model checking to study *automation surprises* (i.e., the system works as designed

but the pilot is unable to predict or explain the behavior of the aircraft). In [14] the PVS theorem prover and the NuSMV model checker are used to find the potential source of automation surprises in a flight guidance system. In contrast to our work, the work in [13,14] does not deal with multitasking, and [14] does not focus on the cognitive aspects of human behavior.

We discuss the differences with the formal cognitive framework proposed in [7] in the introduction.

Finally, as mentioned in Sect. 2, in [5] we propose a task switching algorithm for non-structured tasks that we extend in the current paper. That work does not provide a formal model, but is used to demonstrate the agreement of our modeling approach with relevant psychological literature.

## 7 Concluding Remarks

In this paper we have presented for the first time a formal executable framework for safety-critical human multitasking. The framework enables the simulation and model checking in Real-Time Maude of a person concurrently interacting with multiple devices of different degrees of safety-criticality. Task switching is modeled through a task ranking procedure which is consistent with studies in psychology. We have shown how Real-Time Maude can be used to automatically analyze prototypical properties in safety-critical human multitasking, and have illustrated our framework with a simple example.

As part of future work, we will in the near future perform experiments in collaboration with psychologists to refine our model. We should also apply our framework on real safety-critical case studies.

**Acknowledgments.** This work has been supported by the project “Metodologie informatiche avanzate per l’analisi di dati biomedici” funded by the University of Pisa (PRA 2017 44).

## References

1. Arrington, C.M., Logan, G.D.: Voluntary task switching: chasing the elusive homunculus. *J. Exp. Psychol. Learn. Mem. Cogn.* **31**(4), 683–702 (2005)
2. Australian Transport Safety Bureau: Dangerous distraction. Safety Investigation Report B2004/0324 (2005)
3. Barrouillet, P., Bernardin, S., Camos, V.: Time constraints and resource sharing in adults’ working memory spans. *J. Exp. Psychol. Gen.* **133**(1), 83–100 (2004)
4. Broccia, G., Milazzo, P., Ölveczky, P.: An executable formal framework for safety-critical human multitasking (2017). Report: <http://www.di.unipi.it/msvbio/software/HumanMultitasking.html>
5. Broccia, G., Milazzo, P., Ölveczky, P.C.: An algorithm for simulating human selective attention. In: Cerone, A., Roveri, M. (eds.) SEFM 2017. LNCS, vol. 10729, pp. 48–55. Springer, Cham (2018). [https://doi.org/10.1007/978-3-319-74781-1\\_4](https://doi.org/10.1007/978-3-319-74781-1_4)
6. Byrne, M.D., Kirlik, A.: Using computational cognitive modeling to diagnose possible sources of aviation error. *Int. J. Aviat. Psychol.* **15**(2), 135–155 (2005)



7. Cerone, A.: A cognitive framework based on rewriting logic for the analysis of interactive systems. In: De Nicola, R., Kühn, E. (eds.) SEFM 2016. LNCS, vol. 9763, pp. 287–303. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-41591-8\\_20](https://doi.org/10.1007/978-3-319-41591-8_20)
8. Clark, T., et al.: Impact of clinical alarms on patient safety. Technical report, ACCE Healthcare Technology Foundation (2006)
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). <https://doi.org/10.1007/978-3-540-71999-1>
10. Dingus, T.A., Guo, F., Lee, S., Antin, J.F., Perez, M., Buchanan-King, M., Hankey, J.: Driver crash risk factors and prevalence evaluation using naturalistic driving data. *Proc. Nat. Acad. Sci.* **113**(10), 2636–2641 (2016)
11. Dismukes, R., Nowinski, J.: Prospective memory, concurrent task management, and pilot error. In: *Attention: From Theory to Practice*. Oxford University Press, Oxford (2007)
12. de Fockert, J.W., Rees, G., Frith, C.D., Lavie, N.: The role of working memory in visual selective attention. *Science* **291**(5509), 1803–1806 (2001)
13. Gelman, G., Feigh, K.M., Rushby, J.M.: Example of a complementary use of model checking and human performance simulation. *IEEE Trans. Hum.-Mach. Syst.* **44**(5), 576–590 (2014)
14. Joshi, A., Miller, S.P., Heimdahl, M.P.E.: Mode confusion analysis of a flight guidance system using formal methods. In: *Digital Avionics Systems Conference (DASC 2003)*. IEEE (2003)
15. Lepri, D., Abraham, E., Ölveczky, P.C.: Sound and complete timed CTL model checking of timed Kripke structures and real-time rewrite theories. *Sci. Comput. Program.* **99**, 128–192 (2015)
16. Lofsky, A.S.: Turn your alarms on! *APSF Newsl.: Off. J. Anesth. Patient Saf. Found.* **19**(4), 43 (2005)
17. Miller, G.A.: The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychol. Rev.* **63**(2), 81–97 (1956)
18. Ölveczky, P.C.: Real-Time Maude and its applications. In: Escobar, S. (ed.) *WRLA 2014*. LNCS, vol. 8663, pp. 42–79. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-12904-4\\_3](https://doi.org/10.1007/978-3-319-12904-4_3)
19. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. *High-Order Symb. Comput.* **20**(1–2), 161–196 (2007)
20. Salvucci, D.D.: Predicting the effects of in-car interface use on driver performance: an integrated model approach. *Int. J. Hum. Comput. Stud.* **55**(1), 85–107 (2001)
21. Salvucci, D.D., Taatgen, N.A.: Threaded cognition: an integrated theory of concurrent multitasking. *Psychol. Rev.* **115**(1), 101–130 (2008)
22. Wickens, C.D., Gutzwiller, R.S.: The status of the strategic task overload model (STOM) for predicting multi-task management. In: *Proceedings of Human Factors and Ergonomics Society Annual Meeting*, vol. 61, pp. 757–761. SAGE Publications (2017)
23. Wickens, C.D., Sebok, A., Li, H., Sarter, N., Gacy, A.M.: Using modeling and simulation to predict operator performance and automation-induced complacency with robotic automation: a case study and empirical validation. *Hum. Fact.* **57**(6), 959–975 (2015)
24. Wickens, C.D., Gutzwiller, R.S., Vieane, A., Clegg, B.A., Sebok, A., Janes, J.: Time sharing between robotics and process control: validating a model of attention switching. *Hum. Fact.* **58**(2), 322–343 (2016)



# Simpler Specifications and Easier Proofs of Distributed Algorithms Using History Variables

Saksham Chand<sup>(✉)</sup> and Yanhong A. Liu

Stony Brook University, Stony Brook, NY 11794, USA  
{schand,liu}@cs.stonybrook.edu

**Abstract.** This paper studies specifications and proofs of distributed algorithms when only message history variables are used, using Basic Paxos and Multi-Paxos for distributed consensus as precise case studies. We show that not using and maintaining other state variables yields simpler specifications that are more declarative and easier to understand. It also allows easier proofs to be developed by needing fewer invariants and facilitating proof derivations. Furthermore, the proofs are mechanically checked more efficiently.

We show that specifications in  $TLA^+$  and proofs in  $TLA^+$  Proof System (TLAPS) are reduced by 25% and 27%, respectively, for Basic Paxos, and 46% (from about 100 lines to about 50 lines) and 48% (from about 1000 lines to about 500 lines), respectively, for Multi-Paxos. Overall we need 54% fewer manually written invariants and our proofs have 46% fewer obligations. Our proof for Basic Paxos takes 26% less time than Lamport et al.'s for TLAPS to check, and our proofs for Multi-Paxos are checked by TLAPS within 1.5 min whereas prior proofs for Multi-Paxos fail to be checked in the new version of TLAPS.

## 1 Introduction

Reasoning about correctness of distributed algorithms is notoriously difficult due to a number of reasons including concurrency, asynchronous networks, unbounded delay, and arbitrary failures. Emerging technologies like autonomous cars are bringing vehicular clouds closer to reality [9], decentralized digital currencies are gathering more attention from academia and industry than ever [31], and with the explosion in the number of nano- and pico- satellites being launched, a similar trend is expected in the field of space exploration as well [29]. All of these systems deal with critical resources like human life, currency, and intricate machinery. This only amplifies the need for employing formal methods to guarantee their correctness.

---

This work was supported in part by NSF grants CCF-1414078 and CCF-1248184 and ONR grant N000141512208. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

Verification of distributed algorithms continues to pose a demanding challenge to computer scientists, exacerbated by the fact that paper proofs of these algorithms cannot be trusted [33]. The usual line of reasoning in static analysis of such systems involves manually writing invariants and then using theorem provers to verify that the invariants follow from the specification and that they imply correctness.

A distributed system comprises a set of processes communicating with each other by message passing while performing local actions that may be triggered upon receiving a set of messages and may conclude with sending a set of messages [14, 15]. As such, data processed by any distributed process fall into two categories: (i) *History Variables*: Sets of all messages sent and received<sup>1</sup> and (ii) *Derived Variables*: Local data maintained for efficient computation. Derived variables are often used to maintain results of aggregate queries over sent and received messages.

While reading and writing pseudocode, derived variables are helpful because instead of writing the definition of the variable everywhere, the variable is used instead. Human readers would recall the definition and convince themselves how the algorithm works. While this approach works well for humans, the same is not true for provers. For specifications written with derived variables, invariants have to be added to their proofs which, at the very least, establish that the derived variable implements its definition.

One reason to use derived variables in formal specifications is their existence in pseudocode. Another reason is the lack of high-level languages that provide elegant support for quantifications, history variables, and automatic optimal maintenance of aggregate queries over history variables. The barrier of lack of executable language support for such richness is overcome by high-level languages like DistAlgo [21], which provides native support for history variables, quantifications, and aggregate queries. This motivated us to dispense with derived variables, and study specifications written with only history variables and the impact of this change on their proofs.

Note that uses of history variables provide higher-level specifications of systems in terms of what to compute, as opposed to how to compute with employing and updating derived variables. It makes proofs easier, independent of the logics used for doing the proofs, because important invariants are captured directly in the specifications, rather than hidden under all the incremental updates. On the other hand, it can make model checking much less efficient, just as it can make straightforward execution much less efficient. This is not only because high-level queries are time consuming, but also because maintaining history variables can blow up the state space. This is why automatic incrementalization [10, 23, 27, 28] is essential for efficient implementations, including implementations of distributed algorithms [20, 22]. The same transformations for incrementalization can drastically speed up both program execution and model checking.

---

<sup>1</sup> This is different than some other references of the term history variables that include sequences of local actions, i.e., execution history [6].

*Contributions.* We first describe a systematic style to write specifications of distributed algorithms using message history variables. The only variables in these specifications are the sets of sent and received messages. We show (i) how these are different from the usual pseudocode, (ii) why these are sufficient for specifying all distributed algorithms, and (iii) when these are better for the provers than other specifications. A method is then explained which, given such specifications, allows us to systematically derive many important invariants which are needed to prove correctness. This method exploits the monotonic increase of the sets of sent and received messages—messages can only be added or read from these sets, not updated or deleted.

We use three existing specifications and their Safety proofs as our case studies: (i) Basic Paxos for single-valued consensus by Lamport et al., distributed as an example with the TLA<sup>+</sup> Proof System (TLAPS) [19], (ii) Multi-Paxos for multi-valued consensus [2], and (iii) Multi-Paxos with preemption [2]. Paxos is chosen because it is famous for being a difficult algorithm to grasp, while at the same time it is the core algorithm for distributed consensus—the most fundamental problem in distributed computing. We show that our approach led to significantly reduced sizes of specifications and proofs, numbers of needed manually written invariants, and proof checking times. Our specifications and proofs are available at <https://github.com/sachand/HistVar>.

*Paper Overview.* Section 2 details our style of writing specifications using Basic Paxos as an example. We then describe our strategy to systematically derive invariants in Sect. 3 while also showing how using history variables leads to needing fewer invariants. We discuss Multi-Paxos briefly in Sect. 4. Results comparing our specifications and proofs with existing work is detailed in Sect. 5. Section 6 concludes with related work.

## 2 Specifications Using Message History Variables

We demonstrate our approach by developing a specification of Basic Paxos in which we only maintain the set of sent messages. This specification is made to correspond to the specification of Basic Paxos in TLA<sup>+</sup> written by Lamport et al. [19]. This is done intentionally to better understand the applicability of our approach. We also simultaneously show Lamport’s description of the algorithm in English [17] to aid the comparison, except we rename message types and variable names to match those in his TLA<sup>+</sup> specification: *prepare* and *accept* messages are renamed **1a** and **2a** respectively, their responses are renamed **1b** and **2b**, respectively, and variable *n* is renamed *b* and *bal* in different places.

**Distributed consensus.** The basic consensus problem, called single-value consensus or single-decree consensus, is to ensure that a single value is chosen from among the values proposed by the processes. The safety requirements for consensus are [17]:

- Only a value that has been proposed may be chosen.
- Only a single value is chosen.

This is formally defined as

$$\text{Safety} \triangleq \forall v1, v2 \in \mathcal{V} : \phi(v1) \wedge \phi(v2) \Rightarrow v1 = v2 \quad (1)$$

where  $\mathcal{V}$  is the set of possible proposed values, and  $\phi$  is a predicate that given a value  $v$  evaluates to true iff  $v$  was chosen by the algorithm. The specification of  $\phi$  is part of the algorithm.

**Basic Paxos.** Paxos solves the problem of consensus. Two main roles of the algorithm are performed by two kinds of processes:

- $\mathcal{P}$  is the set of proposers. These processes propose values that can be chosen.
- $\mathcal{A}$  is the set of acceptors. These processes vote for proposed values. A value is chosen when there are enough votes for it.

A set  $\mathcal{Q}$  of subsets of the acceptors, that is  $\mathcal{Q} \subseteq 2^{\mathcal{A}}$ , is used as a quorum system. It must satisfy the following properties:

- $\mathcal{Q}$  is a set cover for  $\mathcal{A}$ — $\bigcup_{Q \in \mathcal{Q}} Q = \mathcal{A}$ .
- Any two quorums overlap— $\forall Q1, Q2 \in \mathcal{Q} : Q1 \cap Q2 \neq \emptyset$ .

The most commonly used quorum system takes any majority of acceptors as an element in  $\mathcal{Q}$ . For e.g., if  $\mathcal{A} = \{1, 2, 3\}$ , then the majority based quorum set is  $\mathcal{Q} = \{\{1, 2\}, \{2, 3\}, \{1, 3\}, \{1, 2, 3\}\}$ . Quorums are needed because the system can have arbitrary failures. If a process waits for replies from all other processes, as in Two-Phase Commit, the system will hang in the presence of even one failed process. In the mentioned example, the system will continue to work even if acceptor 3 fails because at least one quorum, which is  $\{1, 2\}$ , is alive.

Basic Paxos solves the problem of single-value consensus. It defines predicate  $\phi$  as

$$\phi(v) \triangleq \exists Q \in \mathcal{Q} : \forall a \in Q : \exists b \in \mathcal{B} : \text{sent}(\text{“2b”}, a, b, v) \quad (2)$$

where  $\mathcal{B}$  is the set of proposal numbers, also called ballot numbers, which is any set that can be strictly totally ordered.  $\text{sent}(\text{“2b”}, a, b, v)$  means that a message of type **2b** with ballot number  $b$  and value  $v$  was sent by acceptor  $a$  (to some set of processes). An acceptor votes by sending such a message.

**Variables.** Lamport et al.’s specification of Basic Paxos has four global variables.

- *msgs*—history variable maintaining the set of messages that have been sent. Processes read from or add to this set but cannot remove from it. We rename this to *sent* in both ours and Lamport et al.’s specifications for clarity purposes. This is the only variable maintained in our specifications.
- *maxBal*—for each acceptor, the highest ballot seen by it.

<b>Phase 1a.</b> A proposer selects a proposal number $b$ and sends a <b>1a</b> request with number $b$ to a majority of acceptors.	
Lamport et al.'s	Using <i>sent</i> only
$Phase1a(b \in \mathcal{B}) \triangleq$ $\nexists m \in sent : (m.type = \text{"1a"}) \wedge (m.bal = b)$ $\wedge Send([type \mapsto \text{"1a"}, bal \mapsto b])$ $\wedge UNCHANGED \langle maxVBal, maxBal, maxVal \rangle$	$Phase1a(b \in \mathcal{B}) \triangleq$ $Send([type \mapsto \text{"1a"}, bal \mapsto b])$

**Fig. 1.** Phase 1a of Basic Paxos

<b>Phase 1b.</b> If an acceptor receives a <b>1a</b> request with number $bal$ greater than that of any <b>1a</b> request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than $bal$ and with the highest-numbered proposal (if any) that it has accepted.	
Lamport et al.'s	Using <i>sent</i> only
$Phase1b(a \in \mathcal{A}) \triangleq$ $\exists m \in sent :$ $\wedge m.type = \text{"1a"}$ $\wedge m.bal > maxBal[a]$ $\wedge Send([type \mapsto \text{"1b"},$ $acc \mapsto a, bal \mapsto m.bal,$ $maxVBal \mapsto maxVBal[a],$ $maxVal \mapsto maxVal[a]])$ $\wedge maxBal' =$ $[maxBal \text{ EXCEPT } ![a] = m.bal]$ $\wedge UNCHANGED \langle maxVBal, maxVal \rangle$	$Phase1b(a \in \mathcal{A}) \triangleq$ $\exists m \in sent, r \in max\_prop(a) :$ $\wedge m.type = \text{"1a"}$ $\wedge \forall m2 \in sent : m2.type \in \{\text{"1b"}, \text{"2b"}\} \wedge$ $m2.acc = a \Rightarrow m.bal > m2.bal$ $\wedge Send([type \mapsto \text{"1b"},$ $acc \mapsto a, bal \mapsto m.bal,$ $maxVBal \mapsto r.bal,$ $maxVal \mapsto r.val])$ $2bs(a) \triangleq \{m \in sent : m.type = \text{"2b"} \wedge m.acc = a\}$ $max\_prop(a) \triangleq$ $\text{IF } 2bs(a) = \emptyset \text{ THEN } \{[bal \mapsto -1, val \mapsto \perp]\}$ $\text{ELSE } \{m \in 2bs(a) : \forall m2 \in 2bs(a) : m.bal \geq m2.bal\}$

**Fig. 2.** Phase 1b of Basic Paxos

- $maxVBal$  and  $maxVal$ —for each acceptor,  $maxVBal$  is the highest ballot in which it has voted, and  $maxVal$  is the value it voted for in that ballot.

**Algorithm steps.** The algorithm consists of repeatedly executing two phases. Each phase comprises two actions, one by acceptors and one by proposers.

- **Phase 1a.** Figure 1 shows Lamport's description in English followed by Lamport et al.'s and our specifications in TLA<sup>+</sup>. *Send* is an operator that adds its argument to *sent*, i.e.,  $Send(m) \triangleq sent' = sent \cup \{m\}$ .
  1. The first conjunct in Lamport et al.'s specification is not mentioned in the English description and is not needed. Therefore it was removed.
  2. The third conjunct is also removed because the only variable our specification maintains is *sent*, which is updated by *Send*.

Lamport et. al's	Using <i>sent</i> only
<p><b>Phase 2a.</b> If the proposer receives a response to its <b>1a</b> requests (numbered <math>b</math>) from a majority of acceptors, then it sends a <b>2a</b> request to each of those acceptors for a proposal numbered <math>b</math> with a value <math>v</math>, where <math>v</math> is the value of the highest-numbered proposal among the <b>1b</b> responses, or is any value if the responses reported no proposals.</p> <p><math>Phase2a(b \in \mathcal{B}) \triangleq</math></p> $\begin{aligned} & \wedge \nexists m \in sent : m.type = \text{"2a"} \wedge m.bal = b \\ & \wedge \exists v \in \mathcal{V}, Q \in \mathcal{Q}, S \subseteq \{m \in sent : \\ & \quad m.type = \text{"1b"} \wedge m.bal = b\} : \\ & \quad \wedge \forall a \in Q : \exists m \in S : m.acc = a \\ & \quad \wedge \forall m \in S : m.maxVBal = -1 \\ & \quad \vee \exists c \in 0..(b-1) : \\ & \quad \quad \wedge \forall m \in S : m.maxVBal \leq c \\ & \quad \quad \wedge \exists m \in S : \wedge m.maxVBal = c \\ & \quad \quad \quad \wedge m.maxVal = v \\ & \wedge Send([type \mapsto \text{"2a"}, bal \mapsto b, val \mapsto v]) \\ & \wedge \text{UNCHANGED} \langle maxBal, maxVBal, \\ & \quad maxVal \rangle \end{aligned}$	<p><math>Phase2a(b \in \mathcal{B}) \triangleq</math></p> $\begin{aligned} & \wedge \nexists m \in sent : m.type = \text{"2a"} \wedge m.bal = b \\ & \wedge \exists v \in \mathcal{V}, Q \in \mathcal{Q}, S \subseteq \{m \in sent : \\ & \quad m.type = \text{"1b"} \wedge m.bal = b\} : \\ & \quad \wedge \forall a \in Q : \exists m \in S : m.acc = a \\ & \quad \wedge \forall m \in S : m.maxVBal = -1 \\ & \quad \vee \exists c \in 0..(b-1) : \\ & \quad \quad \wedge \forall m \in S : m.maxVBal \leq c \\ & \quad \quad \wedge \exists m \in S : \wedge m.maxVBal = c \\ & \quad \quad \quad \wedge m.maxVal = v \\ & \wedge Send([type \mapsto \text{"2a"}, bal \mapsto b, val \mapsto v]) \end{aligned}$

**Fig. 3.** Phase 2a of Basic Paxos

- **Phase 1b.** Figure 2 shows the English description and the specifications of Phase 1b. The first two conjuncts in both specifications capture the precondition in the English description. The remaining conjuncts specify the action.
  1. The first conjunct states that message  $m$  received by acceptor  $a$  is of type **1a**.
  2. The second conjunct ensures that the proposal number  $bal$  in the **1a** message  $m$  is higher than that of any **1a** request responded to by  $a$ . In Lamport et al.'s specification, derived variable  $maxBal[a]$  maintains the highest proposal number that  $a$  has ever responded to, in **1b** and **2b** messages, and its second conjunct uses  $m.bal > maxBal[a]$ . Using *sent* only, we capture this intent more directly, as  $\forall m2 \in sent : m2.type \in \{\text{"1b"}, \text{"2b"}\} \wedge m2.acc = a \Rightarrow m.bal > m2.bal$ , because those  $m2$ 's are the response messages that  $a$  has ever sent.
  3. The third conjunct is the action of sending a promise (**1b** message) not to accept any more proposals numbered less than  $bal$  and with the highest-numbered proposal (if any) that  $a$  has accepted, i.e., has sent a **2b** message. This proposal is maintained in Lamport et al.'s specification in derived variables  $maxVBal$  and  $maxVal$ . We specify this proposal as  $max\_prop(a)$ , which is either the set of proposals that have the highest proposal number among all accepted by  $a$  or if  $a$  has not accepted anything, then  $\{[bal \mapsto -1, val \mapsto \perp]\}$  as the default, where  $-1 \notin \mathcal{B}$  and is smaller than all ballots and  $\perp \notin \mathcal{V}$ . This corresponds to initialization in Lamport et al.'s specification as shown in Fig. 5.
  4. The remaining conjuncts in Lamport et al.'s specification maintain the variable  $maxBal[a]$ . A compiler that implements incrementalization [23] over queries would automatically generate and maintain such a derived variable to optimize the corresponding query.

- **Phase 2a.** Figure 3 shows Phase 2a. The specifications differ from the English description by using a set of quorums,  $\mathcal{Q}$ , instead of a majority. The only difference between the two specifications is the removed **UNCHANGED** conjunct when using *sent* only. It is important to note that the English description fails to mention the first conjunct—a conjunct without which the specification is unsafe. Every **2a** message must have a unique ballot. Note that the first conjunct in Lamport et. al.’s specification (and therefore ours as well) states that none of the **2a** messages sent so far has *bal* equal to *b*. This is not directly implementable in a real system because this quantification query requires accessing message histories of all processes. We leave this query as is for two main reasons: (i) The focus of this paper is to demonstrate the use of history variables against derived variables and compare them in the light of simpler specification and verification. This removes derived variables but leaves queries on history variables unchanged even though they are not directly implementable. (ii) There is a commonly-used, straightforward, efficient way to implement this query - namely realizing ballot as a tuple in  $\mathbb{N} \times \mathcal{P}$  [16]. So a proposer only executes Phase 2a on a ballot proposed by itself (i.e., sent a **1a** message with that ballot) and, for efficient implementation, only executes Phase 2a on the highest ballot that it has proposed.
- **Phase 2b.** Figure 4 shows Phase 2b. Like Phase 1b, we replace the second conjunct with the corresponding query over *sent* and remove updates to the derived variables.

<b>Phase 2b.</b> If an acceptor receives an <b>2a</b> request for a proposal numbered <i>bal</i> , it accepts the proposal unless it has already responded to a <b>1a</b> request having a number greater than <i>bal</i> .	
Lamport et al.’s	Using <i>sent</i> only
$Phase2b(a \in \mathcal{A}) \triangleq$	$Phase2b(a \in \mathcal{A}) \triangleq$
$\exists m \in sent :$	$\exists m \in sent :$
$\wedge m.type = \text{“2a”}$	$\wedge m.type = \text{“2a”}$
$\wedge m.bal \geq maxBal[a]$	$\wedge \forall m2 \in sent : m2.type \in \{\text{“1b”}, \text{“2b”}\} \wedge$ $m2.acc = a \Rightarrow m.bal \geq m2.bal$
$\wedge Send([type \mapsto \text{“2b”}, acc \mapsto a,$ $bal \mapsto m.bal, val \mapsto m.val])$	$\wedge Send([type \mapsto \text{“2b”}, acc \mapsto a,$ $bal \mapsto m.bal, val \mapsto m.val])$
$\wedge maxBal' = [maxBal \text{ EXCEPT } ![a] = m.bal]$	
$\wedge maxVbal' = [maxBal \text{ EXCEPT } ![a] = m.bal]$	
$\wedge maxVal' = [maxVal \text{ EXCEPT } ![a] = m.val]$	

Fig. 4. Phase 2b of Basic Paxos

**Complete algorithm specification.** To complete the algorithm specification, we define, and compare, *vars*, *Init*, *Next*, and *Spec* which are typical TLA<sup>+</sup> operator names for the set of variables, the initial state, possible actions leading to the next state, and the system specification, respectively, in Fig. 5.

Lamport et al.’s initialization of *maxVbal* and *maxVal* to  $-1$  and  $\perp$ , respectively, is moved to our definition of *max\_prop* in Fig. 2. We do not need initialization of *maxBal* because if no **1b** or **2b** messages have been sent, the universally



quantified queries over them would be vacuously true. In Lamport et al.’s specification, this is achieved by initializing  $maxBal$  to  $-1$ , which is smaller than all ballots, and thus, the conjunct  $m.bal > maxBal[a]$  in Fig. 2 holds for the first **1a** message received.

Lamport et al.’s	Using <i>sent</i> only
$vars \triangleq \langle sent, maxBal, maxVVal, maxVal \rangle$	$vars \triangleq \langle sent \rangle$
$Init \triangleq \wedge sent = \emptyset$ $\wedge maxVVal = [a \in \mathcal{A} \mapsto -1]$ $\wedge maxBal = [a \in \mathcal{A} \mapsto -1]$ $\wedge maxVal = [a \in \mathcal{A} \mapsto \perp]$	$Init \triangleq sent = \emptyset$
$Next \triangleq \vee \exists b \in \mathcal{B} : Phase1a(b) \vee Phase2a(b)$ $\vee \exists a \in \mathcal{A} : Phase1b(a) \vee Phase2b(a)$	
$Spec \triangleq Init \wedge \square [Next]_{vars}$	

Fig. 5. Complete algorithm specification

### 3 Invariants and Proofs Using Message History Variables

Invariants of a distributed algorithm can be categorized into the following three kinds:

1. *Type invariants.* These ensure that all data processed in the algorithm is of valid type. For example, messages of type **1a** must have a field  $bal \in \mathcal{B}$ . If an action sends a **1a** message with  $bal$  missing or  $bal \notin \mathcal{B}$ , a type invariant is violated.
2. *Message invariants.* These are invariants defined on message history variables. For example, each message of type **2a** has a unique  $bal$ . This is expressed by the invariant  $\forall m1, m2 \in sent : m1.type = \text{“2a”} \wedge m2.type = \text{“2a”} \wedge m1.bal = m2.bal \Rightarrow m1 = m2$ .
3. *Process invariants.* These state properties about the data maintained in derived variables. For example, in Lamport et al.’s specification, one such invariant is that for any acceptor  $a$ ,  $maxBal[a] \geq maxVVal[a]$ .

Figure 6 shows and compares all invariants used in Lamport et al.’s proof vs. ours. The following operators are used in the invariants for brevity:

$$\begin{aligned}
 VotedForIn(a, v, b) &\triangleq \exists m \in sent : \\
 &\quad m.type = \text{“2b”} \wedge m.acc = a \wedge m.val = v \wedge m.bal = b \\
 WontVoteIn(a, b) &\triangleq \forall v \in \mathcal{V} : \neg VotedForIn(a, v, b) \wedge && \text{– Lamport et al.’s} \\
 &\quad maxBal[a] > b \\
 WontVoteIn(a, b) &\triangleq \forall v \in \mathcal{V} : \neg VotedForIn(a, v, b) \wedge && \text{– Using } sent \text{ only} \\
 &\quad \exists m \in sent : m.type \in \{\text{“1b”}, \text{“2b”}\} \wedge m.acc = a \wedge m.bal > b \\
 SafeAt(v, b) &\triangleq \forall b2 \in 0..(b-1) : \exists Q \in \mathcal{Q} : \forall a \in \mathcal{Q} : \\
 &\quad VotedForIn(a, v, b2) \vee WontVoteIn(a, b2)
 \end{aligned} \tag{3}$$

	Lamport et al.'s proof	Our proof
Type Invariants	(I1) $sent \subseteq Messages$ (I2) $maxVBal \in [\mathcal{A} \rightarrow \mathcal{B} \cup \{-1\}]$ (I3) $maxBal \in [\mathcal{A} \rightarrow \mathcal{B} \cup \{-1\}]$ (I4) $maxVal \in [\mathcal{A} \rightarrow \mathcal{V} \cup \{\perp\}]$	$sent \subseteq Messages$
Process Invariants $\forall a \in \mathcal{A}$	(I5) $maxBal[a] \geq maxVBal[a]$ (I6) $maxVal[a] = \perp \Leftrightarrow maxVBal[a] = -1$ (I7) $maxVBal[a] \geq 0 \Rightarrow$ $VotedForIn(a, maxVal[a], maxVBal[a])$ (I8) $\forall b \in \mathcal{B} : b > maxVBal[a] \Rightarrow$ $\nexists v \in \mathcal{V} : VotedForIn(a, v, b)$	
Message Invariants $\forall m \in sent$	(I9) $m.type = "2b" \Rightarrow m.bal \leq maxVBal[m.acc]$ (I10) $m.type = "1b" \Rightarrow m.bal \leq maxBal[m.acc]$	
	(I11) $m.type = "1b" \Rightarrow$ $\vee \wedge m.maxVal \in \mathcal{V} \wedge m.maxVBal \in \mathcal{B}$ $\wedge VotedForIn(m.acc,$ $m.maxVal, m.maxVBal)$ $\vee m.maxVBal = -1 \wedge m.maxVal = \perp$	$m.type = "1b" \Rightarrow$  $\vee VotedForIn(m.acc,$ $m.maxVal, m.maxVBal)$ $\vee m.maxVBal = -1$
	(I12) $m.type = "1b" \Rightarrow$ $\forall b2 \in m.maxVBal + 1..m.bal - 1 : \nexists v \in \mathcal{V} : VotedForIn(m.acc, v, b2)$ (I13) $m.type = "2a" \Rightarrow SafeAt(m.val, m.bal)$ (I14) $m.type = "2a" \Rightarrow$ $\forall m2 \in sent : m2.type = "2a" \wedge m2.bal = m.bal \Rightarrow m2 = m$ (I15) $m.type = "2b" \Rightarrow$ $\exists m2 \in sent : m2.type = "2a" \wedge m2.bal = m.bal \wedge m2.val = m.val$	

**Fig. 6.** Comparison of invariants. Our proof does not need I2–I10, and needs only I1, a simpler I11, and I12–I15.

**Type invariants reduced to one.** Lamport et al. define four type invariants, one for each variable they maintain. *Messages* is the set of all possible valid messages. We require only one, (I1). This invariant asserts that the type of all sent messages is valid. (I2–4) are not applicable to our specification.

**Process invariants not needed.** Lamport et al. define four process invariants, (I5–8), regarding variables *maxVal*, *maxVBal*, and *maxBal*. They are not applicable to our specification, and need not be given in our proof.

(I5) Because  $maxBal[a]$  is the highest ballot ever seen by  $a$  and  $maxVBal[a]$  is the highest ballot  $a$  has voted for, we have

$$\begin{aligned}
 maxBal[a] &= \max(\{m.bal : m \in sent \wedge m.type \in \{“1b”, “2b”\} \wedge m.acc = a\}) \\
 maxVBal[a] &= \max(\{m.bal : m \in sent \wedge m.type \in \{“2b”\} \wedge m.acc = a\}) \quad (4)
 \end{aligned}$$

where  $\max(S) \triangleq \text{CHOOSE } e \in S \cup \{-1\} : \forall f \in S : e \geq f$ . Note that  $\max$  is not in  $TLA^+$  and has to be user-defined. Invariant (I5) is needed in Lamport et al.’s proof but not ours because they use derived variables whereas we specify the properties directly. For example, for Lamport et al.’s Phase 1b,

one cannot deduce  $m.bal > maxVBal[a]$  without (I5), whereas in our Phase 1b, definitions of  $2bs$  and  $max\_prop$  along with the second conjunct are enough to deduce it.

(I6) Lamport et al.'s proof needs this invariant to prove (I11). Because the initial values are part of *Init* and are not explicitly present in their Phase 1b, this additional invariant is needed to carry this information along. We include the initial values when specifying the action in Phase 1b and therefore do not need such an invariant.

(I7) This invariant is obvious from the definition of *VotedForIn* in Eq. (3) and property of  $maxVBal$  in Eq. (4). The premise  $maxVBal[a] \geq 0$  is needed by Lamport et al.'s proof to differentiate from the initial value  $-1$  of  $maxVBal[a]$ .

(I8) This states that  $a$  has not voted for any value at a ballot greater than  $maxVBal[a]$ . This invariant need not be manually given in our proofs because it is implied from the definition of  $maxVBal[a]$ .

**Message invariants not needed or more easily proved.** Before detailing the message invariants, we present a systematic method that can derive several useful invariants used by Lamport et al. and thus make the proofs easier. This method is based on the following properties of our specifications and distributed algorithms:

1. *sent* monotonically increases, i.e., the only operations on it are read and add.
2. Message invariants hold for each sent message of some type, i.e., they are of the form  $\forall m \in sent : m.type = \tau \Rightarrow \Phi(m)$ , or more conveniently if we define  $sent_\tau = \{m \in sent : m.type = \tau\}$ , we have  $\forall m \in sent_\tau : \Phi(m)$ .
3.  $sent = \emptyset$  initially, so the message invariants are vacuously true in the initial state of the system.
4. Distributed algorithms usually implement a logical clock for ordering two arbitrary messages. In Paxos, this is done by ballots.

We demonstrate our method by deriving (I15). The method is applied for each message type used in the algorithm. Invariant (I15) is about **2b** messages. We first identify all actions that send **2b** messages and then do the following:

1. **Increment.** **2b** messages are sent in Phase 2b as specified in Fig. 4. We first determine the increment to *sent*,  $\Delta(sent)$ , the new messages sent in Phase 2b. We denote a message in  $\Delta(sent)$  by  $\delta$  for brevity. We have, from Fig. 4,

$$\delta = [type \mapsto \text{"2b"}, acc \mapsto a, bal \mapsto m.bal, val \mapsto m.val] \quad (5)$$

2. **Analyze.** We deduce properties about the messages in  $\Delta(sent)$ . For **2b** messages, we deduce the most straightforward property that connects the contents of messages in  $\Delta(sent)$  with the message  $m$ , from Fig. 4,

$$\phi(\delta) = \exists m \in sent : m.type = \text{"2a"} \wedge \delta.bal = m.bal \wedge \delta.val = m.val \quad (6)$$

3. **Integrate.** Because (i) *sent* monotonically increases, and (ii)  $\phi$  is an existential quantification over *sent*,  $\phi$  holds for all increments to *sent*<sub>2b</sub>. Property (i) means that once the existential quantification in  $\phi$  holds, it holds forever. Integrating both sides of Eq. (6) in the space of 2b messages yields (I15), i.e.,

$$\begin{aligned} \Phi(\text{sent}_{2b}) &= \forall m2 \in \text{sent}_{2b} : \exists m \in \text{sent} : m.\text{type} = \text{"2a"} \wedge \\ &\quad m2.\text{bal} = m.\text{bal} \wedge m2.\text{val} = m.\text{val} \end{aligned} \quad (7)$$

The case for  $\phi$  being universally quantified over *sent* is discussed with invariant (I12).

*Other message invariants.* (I9) and (I10) follow directly from Eq. (4) and need not be manually specified for our proof. We also derive (I11), (I12), and (I14) as describe in the following.

**(I11)** Like (I15), (I11) can also be systematically derived, from our Phase 1b in Fig. 2. This invariant is less obvious when variables *maxVal* and *maxVBal* are explicitly used and updated because (i) they are not updated in the same action that uses them, requiring additional invariants to carry their meaning to the proofs involving the actions that use them, and (ii) it is not immediately clear if these variables are being updated in Lamport et al.’s Phase 2b in Fig. 4 because a 2b message is being sent or because a 2a message was received.

**(I12)** To derive (I11) and (I15), we focused on *where* the contents of the new message come from. For (I12), we analyze *why* those contents were chosen. From our Phase 1b with definitions of *2bs* and *max\_prop* in Fig. 2, we have

$$\begin{aligned} \phi(\delta) &= \\ &\quad \forall \Delta \exists m \in \text{sent} : m.\text{type} = \text{"2b"} \wedge m.\text{acc} = \delta.\text{acc} \\ &\quad \wedge \forall m \in \text{sent} : m.\text{type} = \text{"2b"} \wedge m.\text{acc} = \delta.\text{acc} \Rightarrow \delta.\text{maxVBal} \geq m.\text{bal} \\ &\quad \wedge \forall \nexists m \in \text{sent} : m.\text{type} = \text{"2b"} \wedge m.\text{acc} = \delta.\text{acc} \wedge \delta.\text{maxVBal} = -1 \end{aligned} \quad (8)$$

$\phi$  has two disjuncts—the first has a universal quantification and the second has negated existential, which is universal in disguise. If *sent* is universally quantified, integration like for (I15) is not possible because the quantification only holds *at the time of the action*. As new messages are sent in the future, the universal may become violated.

The key is the phrase *at the time*. One way to work around the universal is to add a time field in each message and update it in every action as a message is sent, like using a logical clock. Then, a property like  $\phi(\delta) = \forall m \in \text{sent}_\tau : \psi(m)$  can be integrated to obtain

$$\Phi(\text{sent}_\tau) = \forall m2 \in \text{sent}_\tau : \forall m \in \text{sent} : m.\text{time} < m2.\text{time} \Rightarrow \psi(m) \quad (9)$$

Because ballots act as the logical clock in Paxos, we do not need to specify a separate logical clock and we can perform the above integration on Eq. (8) to obtain the invariant (I12).

(I14) This invariant is of the form  $\forall m1, m2 \in sent_\tau, t : \psi(m1, t) \wedge \psi(m2, t) \Rightarrow m1 = m2$ . In this case,  $\psi(m, t) \triangleq m.bal = t$ . Deriving invariants like (I14) is nontrivial unless  $\psi$  is already known. In some cases,  $\psi$  can be guessed. The intuition is to look for a universal quantification (or negated existential) in the specification of an action. The ideal case is when the quantification is on the message type being sent in the action. Potential candidates for  $\psi$  may be hidden in such quantifications. Moreover, if message history variables are used, these quantifications are easier to identify.

Starting with a guess of  $\psi$ , we identify the change in the counting measure (cardinality) of the set  $\{t : m \in sent_\tau \wedge \psi(m, t)\}$  along with that of  $sent_\tau$ . In the case of (I14), we look for  $\Delta(|\{m.bal : m \in sent_{2a}\}|)$ . From our Phase 2a in Fig. 3, we have

$$\begin{aligned} \Delta(\{m.bal : m \in sent_{2a}\}) &= \{b\} \\ \phi(\Delta(\{m.bal : m \in sent_{2a}\})) &= \nexists m \in sent : m.type = \text{“2a”} \wedge m.bal = b \end{aligned} \tag{10}$$

Rewriting  $\phi$  as  $\{b\} \not\subseteq \{m.bal : m \in sent_{2a}\}$ , it becomes clear that  $\Delta(|\{m.bal : m \in sent_{2a}\}|) = 1$ . Meanwhile,  $\Delta(|\{m \in sent_{2a}\}|) = 1$ . Because the counting measure increases by the same amount for both, (I14) can be derived safely.

## 4 Multi-Paxos

**Specification.** We have developed new specifications of Multi-Paxos and Multi-Paxos with Preemption that use only message history variables, by removing derived variables from the specifications described in Chand et al. [2]. This is done in a way similar to how we removed derived variables from Lamport et al.’s specification of Basic Paxos.

The most interesting action here was preemption. With preemption, if an acceptor receives a **1a** or **2a** message with *bal* smaller than the highest that it has seen, it responds with a **preempt** message that contains the highest ballot seen by the acceptor. Upon receiving such a message, the receiving proposer would pick a new ballot that is higher than the ballots of all received **preempt** messages.

This is a good opportunity to introduce the other message history variable, *received*, the set of all messages received. It is different from *sent* because a message could be delayed indefinitely before being received, if at all. In [2], derived variable *proBallot* is introduced to maintain the result of this query on received messages. We contrast this with our new specification in Fig. 7. *Receive*(*m*) adds message *m* to *received*, i.e.,  $Receive(m) \triangleq received' = received \cup \{m\}$ .

**Verification.** While we observed a 27% decrease in proof size for Basic Paxos, for Multi-Paxos this decrease was 48%. Apart from the points described in Sect. 3, an important player in this decrease was the removal of operator

Chand et al. [2]	Using <i>sent</i> and <i>received</i>
$NewBallot(c \in \mathcal{B}) \triangleq \text{CHOOSE } b \in \mathcal{B} : b > c \wedge$ $\nexists m \in sent : m.type = \text{"1a"} \wedge m.bal = b$  $Preempt(p \in \mathcal{P}) \triangleq \exists m \in sent :$ $\wedge m.type = \text{"preempt"} \wedge m.to = p$ $\wedge m.bal > proBallot[p]$ $\wedge proBallot' = [proBallot \text{ EXCEPT } ![p] =$ $    NewBallot(m.bal)]$ $\wedge \text{UNCHANGED } \langle sent, aVoted, aBal \rangle$  $Phase1a(p \in \mathcal{P}) \triangleq$ $\wedge \nexists m \in sent : (m.type = \text{"1a"}) \wedge$ $    (m.bal = proBallot[p])$ $\wedge Send([type \mapsto \text{"1a"},$ $    from \mapsto p, bal \mapsto proBallot[p]])$ $\wedge \text{UNCHANGED } \langle aVoted, aBal, proBallot \rangle$	$Phase1a(p \in \mathcal{P}) \triangleq \exists b \in \mathcal{B} :$ $\wedge \vee \exists m \in sent :$ $\wedge m.type = \text{"preempt"} \wedge m.to = p$ $\wedge Receive(m)$ $\wedge \forall m2 \in received' : m2.to = p \wedge$ $    m2.type = \text{"preempt"} \Rightarrow b > m2.bal$ $\vee \wedge \nexists m \in sent : m.type = \text{"1a"} \wedge$ $    m.from = p$ $\wedge \text{UNCHANGED } \langle received \rangle$ $\wedge Send([type \mapsto \text{"1a"}, from \mapsto p, bal \mapsto b])$

Fig. 7. Preemption in Multi-Paxos

*MaxVotedBallotInSlot* from Chand et al.’s specifications. This operator was defined as

$$MaxVotedBallotInSlot(D, s) \triangleq \max(\{d.bal : d \in \{d \in D : d.slot = s\}\})$$

Five lemmas were needed in Chand et al.’s proof to assert basic properties of the operator. For example, lemma *MVBISType* stated that if  $D \subseteq [bal : \mathcal{B}, slot : \mathcal{S}, val : \mathcal{V}]$ , then the result of the operator is in  $\mathcal{B} \cup \{-1\}$ . Removing these lemmas and their proofs alone resulted in a decrease of about 100 lines (about 10%) in proof size.

## 5 Results

Table 1 summarizes the results of our specifications and proofs that use only message history variables, compared with those by Lamport et al. and Chand et al. We observe an improvement of around 25% across all stats for Basic Paxos and a staggering 50% for Multi-Paxos and Multi-Paxos with Preemption. Following, we list some important results:

- The specification size decreased by 13 lines (25%) for Basic Paxos, from 52 lines for Lamport et al.’s specification to 39 lines for ours. For Multi-Paxos, the decrease is 36 lines (46%), from 78 lines for Chand et al.’s to 42 lines for ours, and for Multi-Paxos with Preemption, the decrease is 45 lines (46%), from 97 to 52.
- The total number of manually written invariants decreased by 54% overall—by 9 (60%) from 15 to 6 for Basic Paxos, by 8 (50%) from 16 to 8 for Multi-Paxos, and by 9 (53%) from 17 to 8 for Multi-Paxos with Preemption.

This drastic decrease is because we do not maintain the variables  $maxBal$ ,  $maxVBal$ , and  $maxVal$  as explained in Sect. 3.

- The proof size for Basic Paxos decreased by 83 lines (27%), from 310 to 227. This decrease is attributed to the fact that our specification does not use other state variables besides  $sent$ . This decrease is 468 lines (47%), from 988 to 520, for Multi-Paxos, and is 494 lines (48%), from 1032 to 538 for Multi-Paxos with Preemption.
- Proof by contradiction is used twice in the proof by Lamport et al. and thrice for the proofs in Chand et al. We were able to remove all of these because our specification uses queries as opposed to derived variables. The motive behind removing proofs by contradiction is to have easier to understand constructive proofs.
- The total number of proof obligations decreased by 46% overall—by 57 (24%) from 239 to 182 for Basic Paxos, by 450 (49%) from 918 to 468 for Multi-Paxos, and by 468 (49%) from 959 to 491 for Multi-Paxos with Preemption.
- The proof-checking time decreased by 11 s (26%), from 42 to 31 for Basic Paxos. For Multi-Paxos and Multi-Paxos with Preemption, TLAPS took over 3 min for the proofs in [2] and failed (due to updates in the new version of TLAPS) to check the proofs of about 5 obligations. In contrast, our proofs were able to be checked completely in 1.5 min or less.

## 6 Related Work and Conclusion

*History Variables.* History variables have been at the center of much debate since they were introduced in the early 1970s [5–7]. Owicki and Gries [25] use them in an effort to prove properties of parallel programs, criticized by Lamport in his writings [13]. Contrary to ours, their history variables were ghost or auxiliary variables introduced for the sole purpose of simpler proofs. Our history variables are  $sent$  and  $received$ , whose contents are actually processed in all distributed system implementations.

Recently, Lamport and Merz [18] present rules to add history variables, among other auxiliary variables, to a low-level specification so that a refinement mapping from a high-level one can be established. The idea is to prove invariants in the high-level specification that serves as an abstraction of the low-level specification. In contrast, we focus on high-level specifications because our target executable language is DistAlgo, and efficient lower-level implementations can be generated systematically from high-level code.

*Specification and Verification.* Several systems [4, 8, 30], models [3, 24, 32], and methods [1, 11, 12, 26] have been developed in the past to specify distributed algorithms and mechanically check proofs of the safety and liveness properties of the algorithms. This work is orthogonal to them in the sense that the idea of maintaining only message history variables can be incorporated in their specifications as well.

**Table 1.** Summary of results. Lam is from Lamport et al., Cha is from Chand et al. [2], Us is ours in this paper, and Decr is percentage of decrease by ours. Specification size and proof size are measured in lines. An obligation is a condition that TLAPS checks. The time to check is on an Intel i7-4720HQ 2.6 GHz CPU with 16 GB of memory, running 64-bit Windows 10 Home (v1709 b16299.98) and TLAPS 1.5.4. \*indicates that the new version of TLAPS failed to check the proof and gave up on checking after that number of seconds. †(I10)–(I12) are **1b** invariants, (I13) and (I14) are **2a** invariants, and (I9) and (I15) are **2b** invariants for Basic Paxos.

Metric	Basic Paxos			Multi-Paxos			Multi-Paxos w/ Preemption		
	Lam	Us	Decr	Cha	Us	Decr	Cha	Us	Decr
Spec. size excl. comments	52	39	25%	78	42	46%	97	52	46%
# invariants	15	6	60%	16	8	50%	17	8	53%
# type invariants	4	1	75%	4	1	75%	5	1	80%
# process invariants	4	0	100%	4	0	100%	4	0	100%
# message invariants	7	5	29%	8	7	13%	8	7	13%
Proof size excl. comments	310	227	27%	988	520	47%	1032	538	48%
Type invariants’ proof size	22	21	5%	54	34	37%	75	38	49%
Process invariants’ proof size	27	0	100%	136	0	100%	141	0	100%
<b>1b</b> † invariants’ proof size	21	15	29%	133	70	47%	133	70	47%
<b>2a</b> † invariants’ proof size	73	57	22%	264	120	55%	269	120	55%
<b>2b</b> † invariants’ proof size	14	12	14%	94	73	22%	94	73	22%
# proofs by contradiction	2	0	100%	3	0	100%	3	0	100%
# obligations in TLAPS	239	182	24%	918	468	49%	959	491	49%
Type inv proof obligations	17	17	0%	69	52	25%	100	60	40%
Process inv proof obligations	39	0	100%	163	0	100%	173	0	100%
<b>1b</b> † inv proof obligations	12	10	17%	160	80	50%	160	80	50%
<b>2a</b> † inv proof obligations	62	52	16%	241	145	40%	249	145	42%
<b>2b</b> † inv proof obligations	9	9	0%	77	44	43%	77	44	43%
TLAPS check time (seconds)	42	31	26%	>191*	80	>58%	>208*	90	>57%

Closer to us in terms of the specification is the work by Padon et al. [26], which does not define any variables and instead defines predicate relations which would correspond to manipulations of our history variables. For example,  $Send([type \mapsto \text{“1a”}, bal \mapsto b])$  is denoted by  $start\_round\_msg(b)$ . Instead of using  $TLA^+$ , the temporal logic of actions, they specify Paxos in first-order logic to later exploit benefits of Effectively Propositional Logic, such as satisfiability being decidable in it.

In contrast, we present a method to specify distributed algorithms using history variables, implementable in high-level executable languages like DistAlgo,



and then show (i) how such specifications require fewer invariants for proofs and (ii) how several important invariants can be systematically derived.

## References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. *Theoret. Comput. Sci.* **82**(2), 253–284 (1991)
2. Chand, S., Liu, Y.A., Stoller, S.D.: Formal verification of multi-paxos for distributed consensus. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 119–136. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-48989-6\\_8](https://doi.org/10.1007/978-3-319-48989-6_8)
3. Charron-Bost, B., Schiper, A.: The Heard-Of model: computing in distributed systems with benign faults. *Distrib. Comput.* **22**(1), 49–71 (2009)
4. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: The TLA<sup>+</sup> proof system: building a heterogeneous verification platform. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, p. 44. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-14808-8\\_3](https://doi.org/10.1007/978-3-642-14808-8_3)
5. Clarke, E.M.: Proving correctness of coroutines without history variables. *Acta Inform.* **13**(2), 169–188 (1980)
6. Clint, M.: Program proving: coroutines. *Acta inform.* **2**(1), 50–63 (1973)
7. Clint, M.: On the use of history variables. *Acta Inform.* **16**(1), 15–30 (1981)
8. Drăgoi, C., Henzinger, T.A., Zufferey, D.: PSync: a partially synchronous language for fault-tolerant distributed algorithms. *ACM SIGPLAN Notices* **51**(1), 400–415 (2016)
9. Gerla, M., Lee, E.-K., Pau, G., Lee, U.: Internet of vehicles: from intelligent grid to autonomous cars and vehicular clouds. In: 2014 IEEE World Forum on Internet of Things (WF-IoT), pp. 241–246. IEEE (2014)
10. Gorbovitski, M.: A system for invariant-driven transformations. Ph.D. thesis, Computer Science Department, Stony Brook University (2011)
11. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: IronFleet: proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles, pp. 1–17. ACM (2015)
12. Küfner, P., Nestmann, U., Rickmann, C.: Formal verification of distributed algorithms. In: Baeten, J.C.M., Ball, T., de Boer, F.S. (eds.) TCS 2012. LNCS, vol. 7604, pp. 209–224. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-33475-7\\_15](https://doi.org/10.1007/978-3-642-33475-7_15)
13. Lamport, L.: My writings : proving the correctness of multiprocess programs. <https://lamport.azurewebsites.net/pubs/pubs.html>. Accessed 10 Oct 2017
14. Lamport, L.: The implementation of reliable distributed multiprocess systems. *Comput. Netw.* **2**(2), 95–114 (1978)
15. Lamport, L.: The temporal logic of actions. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **16**(3), 872–923 (1994)
16. Lamport, L.: The part-time parliament. *ACM Trans. Comput. Syst. (TOCS)* **16**(2), 133–169 (1998)
17. Lamport, L.: Paxos made simple. *ACM Sigact News* **32**(4), 18–25 (2001)
18. Lamport, L., Merz, S.: Auxiliary variables in TLA<sup>+</sup>. ArXiv e-prints, March 2017
19. Lamport, L., Merz, S., Doligez, D.: Paxos.tla. <https://github.com/tlaplus/v1-tlapm/blob/master/examples/paxos/Paxos.tla>. Accessed 6 Feb 2018

20. Liu, Y.A., Brandvein, J., Stoller, S.D., Lin, B.: Demand-driven incremental object queries. In: Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming (PPDP), pp. 228–241. ACM (2016)
21. Liu, Y.A., Stoller, S.D., Lin, B.: From clarity to efficiency for distributed algorithms. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **39**(3), 12 (2017)
22. Liu, Y.A., Stoller, S.D., Lin, B., Gorbovitski, M.: From clarity to efficiency for distributed algorithms. In: Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 395–410. ACM (2012)
23. Liu, Y.A.: *Systematic Program Design: From Clarity To Efficiency*. Cambridge University Press, Cambridge (2013)
24. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing, pp. 137–151. ACM (1987)
25. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs i. *Acta Inform.* **6**(4), 319–340 (1976)
26. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made EPR: decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.* **1**(OOPSLA), 108 (2017)
27. Paige, R., Koenig, S.: Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **4**(3), 402–454 (1982)
28. Rothamel, T., Liu, Y.A.: Generating incremental implementations of object-set queries. In: Proceedings of the 7th International Conference on Generative Programming and Component Engineering, pp. 55–66. ACM (2008)
29. Schilling, K.: Perspectives for miniaturized, distributed, networked cooperating systems for space exploration. *Robot. Auton. Syst.* **90**, 118–124 (2017)
30. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. *Proc. ACM Program. Lang.* **2**(POPL), 28 (2017)
31. Tschorsch, F., Scheuermann, B.: Bitcoin and beyond: a technical survey on decentralized digital currencies. *IEEE Commun. Surv. Tutor.* **18**(3), 2084–2123 (2016)
32. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: a framework for implementing and formally verifying distributed systems. In: *ACM SIGPLAN Notices*, vol. 50, pp. 357–368. ACM (2015)
33. Zave, P.: Using lightweight modeling to understand Chord. *ACM SIGCOMM Comput. Commun. Rev.* **42**(2), 49–57 (2012)



# Don't Miss the End: Preventing Unsafe End-of-File Comparisons

Charles Zhuo Chen<sup>(✉)</sup> and Werner Dietl

University of Waterloo, Waterloo, Canada  
{z359chen,werner.dietl}@uwaterloo.ca

**Abstract.** Reading from an `InputStream` or `Reader` in Java either returns the read byte/character or `-1` if the end-of-file (EOF) has been reached. To support the additional `-1` as return value, the `read` methods return an `int`. For correct usage, the return value should be compared to `-1` before being converted to `byte` or `char`. If the conversion was performed before the comparison, it can cause a read-until-EOF-loop to either exit prematurely or be stuck in an infinite loop. The SEI CERT Oracle Coding Standard for Java rule FIO08-J “Distinguish between characters or bytes read from a stream and `-1`” describes this issue in detail. This paper presents a type system that prevents unsafe EOF value comparisons statically and is implemented for Java using the Checker Framework. In an evaluation of 35 projects (9 million LOC) it detected 3 defects in production software, 8 bad coding practices, and no false positives. The overall annotation effort is very low. Overrides for the `read` methods needed to be annotated, requiring a total of 44 annotations. Additionally, 3 annotations for fields and method parameters needed to be added. To the best of our knowledge this is the first open source tool to prevent this security issue.

**Keywords:** Software security · Static analysis · Java type system  
CERT rules · Practice

## 1 Introduction

Reading from an input stream is one of the most basic operations for an application and required in many different domains. Java provides methods `InputStream.read()` and `Reader.read()` for reading bytes or characters from an input stream. `InputStream.read()`<sup>1</sup> returns an `int` in the range of 0 to 255, or `-1` if the end-of-file (EOF) was reached. Similarly, `Reader.read()`<sup>2</sup> returns an `int` in the range of 0 to 65535, or `-1` for EOF. These `read` methods return an `int` in order to distinguish the additional `-1` from the maximum `byte/char` value. As the CERT FIO08-J [3] rule describes, one common usage mistake of these `read` methods is the premature conversion of the read `int` to `byte/char`,

<sup>1</sup> See <https://docs.oracle.com/javase/9/docs/api/java/io/InputStream.html#read-->.

<sup>2</sup> See <https://docs.oracle.com/javase/9/docs/api/java/io/Reader.html#read-->.

before comparing with  $-1$ . In Java, `byte` is defined as an 8 bit signed number, `char` is a 16 bit unsigned Unicode character, and `int` is a 32 bit signed number. The narrowing conversion from the returned `int` to `byte/char` makes it impossible to distinguish the maximum `byte/char` value from the EOF value  $-1$ . Figure 1 shows a simple example of this kind of mistake. If the `int` returned by `Reader.read()` is prematurely converted to a `char`, the EOF value  $-1$  is converted to 65535, resulting in an infinite loop. Similarly, if the `int` returned by `InputStream.read()` is prematurely converted to a `byte`, the maximum stream value 255 is converted to  $-1$ , resulting in the loop to exit prematurely.

```

StringBuffer stringBuffer = new StringBuffer();
char c;
while ((c = (char) reader.read()) != -1) {
    stringBuffer.append(c);
}

```

**Fig. 1.** An example of a FIO08-J rule violation, resulting in an infinite loop.

This paper presents the EOF Value Checker, a tool that allows a conversion from the read `int` to `byte/char` only after comparing with  $-1$ , to guarantee the absence of ambiguously converted results. The tool is designed as a pluggable type system for Java and built using the Checker Framework [2]. With the rich type rules and a standard data-flow framework provided by the Checker Framework, this tool is implemented easily through 312 lines of Java code. This tool guarantees that the FIO08-J rule is never violated. In an evaluation of the tool on 35 real world projects (9 million LOC), it found 3 defects and 8 bad coding styles that violate the FIO08-J rule, and there were zero false positives. Only 47 manual annotations were required in this evaluation. To the best of our knowledge, the EOF Value Checker is the first open source tool that prevents this vulnerability. It is available freely on GitHub<sup>3</sup>.

The rest of this paper is organized as follows. Section 2 presents the EOF Value Checker type system, Sect. 3 presents the implementation of this type system, Sect. 4 presents the case study of applying the EOF Value Checker to 35 open source projects, and Sect. 5 reviews related work. Finally, Sect. 6 concludes.

## 2 Type System

This section presents a qualifier-based refinement type system that guarantees that premature conversions from read `int` to `byte/char` never happen at run time. Section 2.1 introduces the qualifiers and the qualifier hierarchy; Sect. 2.2 explains the type rules; Sect. 2.3 explains default qualifiers; and Sect. 2.4 explains data-flow-sensitive qualifier refinement.

<sup>3</sup> See <https://github.com/oppop/ReadChecker>.

## 2.1 Type Qualifiers and Qualifier Hierarchy

The EOF Value Checker type system provides three qualifiers: `@UnsafeRead`, `@UnknownSafety`, and `@SafeRead`:

- `@UnsafeRead` qualifies `int` types that represent a `byte/char` or the EOF value `-1` **before** being checked against `-1`.
- `@SafeRead` qualifies `int` types that represent a `byte/char` **after** being checked against `-1`.
- `@UnknownSafety` qualifies `int` types without any compile time information about their representations.

All three qualifiers are only meaningful for `int` types. All other types can essentially ignore qualifiers.

The type qualifiers form a simple subtype hierarchy. Figure 2 illustrates the subtyping among type qualifiers.

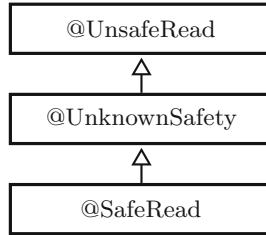


Fig. 2. Qualifier hierarchy of the EOF Value Checker.

## 2.2 Type Casting Rules

The EOF Value Checker restricts the standard type rules for narrowing casts, as shown in Fig. 3. Casts from `@UnsafeRead` to `byte/char` are forbidden and only `@UnknownSafety int` and `@SafeRead int` can be cast to `byte/char`. With the return type of the `read` methods annotated with `@UnsafeRead`, the type cast rules ensure that the read `int` is compared against `-1` before being cast to `byte/char`.

$$\frac{\Gamma \vdash e : Q \text{ int} \quad Q \neq @UnsafeRead}{\Gamma \vdash (\text{byte}) e : Q \text{ byte}} \quad \frac{\Gamma \vdash e : Q \text{ int} \quad Q \neq @UnsafeRead}{\Gamma \vdash (\text{char}) e : Q \text{ char}}$$

Fig. 3. Type rules for narrowing casts. Only casts from `@UnknownSafety` and `@SafeRead` are allowed. Casts from `@UnsafeRead` are forbidden. All other type rules are standard for a pluggable type system and enforce subtype consistency between types.

```
error: @UnsafeRead int should not be casted to char.
line 3: while ((c = (char) reader.read()) != -1) {
                ^
```

**Fig. 4.** The type cast error issued for the example from Fig. 1.

These type cast rules will effectively prevent the read mistake by preventing premature type casts. Figure 4 shows the type cast error issued by the EOF Value Checker for the example shown in Fig. 1.

This type error can be fixed by comparing the read `int` with the EOF value before casting it to `char`. Figure 5 shows the corrected source code.

```
StringBuffer stringBuffer = new StringBuffer();
int data;
while ((data = reader.read()) != -1) {
    stringBuffer.append((char) data);
}
```

**Fig. 5.** A fix of the type cast error shown in Fig. 1.

Note that the cast to `char` is allowed after the comparison against `-1`. The data-flow refinement, explained in Sect. 2.4, refines the type of `data` from `@UnsafeRead` to `@SafeRead` after the `-1` comparison, allowing the cast in the loop body. This fixes the premature conversion without requiring any explicit annotations in the source code.

### 2.3 Default Qualifiers

The type system uses default qualifiers for all type uses, minimizing the manual annotation effort. Defaulting follows the CLIMB-to-top approach from the Checker Framework<sup>4</sup>. Local variables are defaulted with the top qualifier, `@UnsafeRead`, because their effective type will be determined with data-flow-sensitive type refinement.

`@UnknownSafety` is the default qualifier for all other type use locations. Since `@UnknownSafety` is a subtype of `@UnsafeRead`, this means an `@UnsafeRead int` cannot be assigned to a field or passed to a method without explicit `@UnsafeRead` annotation on the field/method parameter. This ensures that an `@UnsafeRead int` isn't lost through a non-local data flow.

Our case study finds that most read `int` are used locally. In very few cases programs assign a read `int` to a field or pass them as a method parameter. In the evaluation of 35 projects, only 1 project requires annotating 2 fields and 1 method parameter with `@UnsafeRead`.

<sup>4</sup> <https://checkerframework.org/manual/#climb-to-top>.

## 2.4 Data-Flow-Sensitive Type Refinement

The EOF Value Checker performs data-flow-sensitive type refinement to minimize the annotation effort. An `@UnsafeRead int` can be refined to `@SafeRead` if the possible run-time values of this `int` are guaranteed to not include `-1`. Correct programs use range checks or comparisons against `-1` to ensure a conversion to `byte/char` is safe. These value comparisons provide static information which the EOF Value Checker can use to ensure casts are safe. The EOF Value Checker applies additional transfer functions on binary comparison nodes in the control-flow graph to refine types.

For a binary comparison node, if one of the operands is `@UnsafeRead int` and the other operand is a constant value, the corresponding transfer function refines the `@UnsafeRead int` to `@SafeRead` in the branch that ensures `-1` is not a possible run-time value of the `@UnsafeRead int`. Figure 6 gives several examples of the data-flow-sensitive refinement of binary comparison nodes in the EOF Value Checker.

```
@UnsafeRead int data = in.read();

// Explicitly compare read result with EOF value -1.
if (data != -1) { /* refines data to @SafeRead */ }

// Only non-EOF values can flow into the block.
if (data == '<' || data == '>') { /* refines data to @SafeRead */ }

// A range check which excludes the EOF value.
if (data >= 0) { /* refines data to @SafeRead */ }
```

**Fig. 6.** Some examples of data-flow-sensitive refinement.

The EOF Value Checker does not perform any constant propagation and only comparisons between `@UnsafeRead int` and literals are refined. This can cause a false positive if an `@UnsafeRead int` is compared with a variable, for which a constant propagation could determine a value. Figure 7 gives an example of this

```
@UnsafeRead int data = in.read();
final int MINUS1 = -1;

// Transfer functions would not refine data to @SafeRead,
// as MINUS1 is not a literal.
if (data != MINUS1) {
    char c = (char) data; // A false positive warning.
}
```

**Fig. 7.** A possible false positive warning due to the absence of constant propagation.

kind of false positive. This could easily be improved by also applying a constant propagation. However, in the evaluation on 35 projects, no false positives are generated, and therefore there are no cases where a constant propagation would help.

### 3 Implementation

The EOF Value Checker type system described in Sect. 2 is implemented as a pluggable type system using the Checker Framework [2].

The type system is independent of the specific stream API. It can be instantiated by annotating methods that need protection against premature conversion as returning `@UnsafeRead`. The EOF Value Checker provides 32 `@UnsafeRead` annotations for the `read` methods in the `InputStream` and `Reader` classes and their subclasses in `java.io`, `java.net`, `javax.swing`, `javax.sound.sampled`, `javax.imageio`, `java.util.zip`, and `java.security` packages. It is easy to provide additional annotations for other APIs.

Overall the implementation effort is very low, totaling 312 lines of Java code. The EOF Value Checker uses the standard type rules and data-flow-sensitive type refinement from the Checker Framework. Only the type rule for casts has been extended as shown in Fig. 3. Only three transfer functions on binary comparison nodes are extended to achieve the data-flow-sensitive type refinement described in Sect. 2.4.

### 4 Experiments

We evaluate the EOF Value Checker on 35 open source projects. The largest project is Apache TomEE, a lightweight JavaEE Application server framework. The other 34 projects are from Apache Commons, a collection of reusable components in wide use. For each project, the EOF Value Checker is ran on the Java source files with a configuration extracted from the project build file. Figure 8 presents the experimental results.

For every resulting warning, we manually identify whether it is a real defect, a bad coding practice, or a false positive. A real defect is to use the prematurely converted result in a comparison to the EOF value, which might lead the reading loop to exit prematurely or to be stuck in an infinite loop. We categorized a warning as a bad coding practice if the prematurely converted `int` is used before the `int` is compared to the EOF value, which can lead to invalid output.

Overall, the EOF Value Checker finds 3 defects in Apache TomEE and Apache Commons IO and 8 bad coding practices in 3 projects. No false positives are generated for all 35 projects. The 3 defects have been reported to the respective project maintainers. The two issues in Apache TomEE have since been fixed.

The overall annotation effort is very low. Overrides for the `read` methods need to be annotated, requiring a total of 44 annotations. Only 1 project needs additional `@UnsafeRead` annotations on 2 fields and 1 method parameter.



Running the EOF Value Checker adds compile time overhead. For the largest project, the EOF Value Checker adds 2.9 times the original compile time as overhead. On average 2.75 times overhead is added. This overhead is expected for a Checker Framework based type system. Future performance improvements to the Checker Framework will also benefit the EOF Value Checker.

Project	Java LOC	Manual Annotations	Bad Style	Defects	Time Overhead
Apache TomEE	1178k	2	2	2	2.9
Apache Commons IO	42k	12	0	1	4.4
Apache Commons BCEL	366k	1	4	0	2.5
Apache Commons Imaging	49k	6	2	0	3.9
Apache Commons Compress	57k	15	0	0	2.9
Apache Commons CSV	9k	2	0	0	1.9
Apache Commons Fileupload	8k	1	0	0	0.9
Apache Commons Net	34k	4	0	0	1.9
Apache Commons VFS	39k	4	0	0	3.6

**Fig. 8.** Case study results. Only projects that have defects, bad coding practices, or explicit annotations are listed. The time overhead is relative to the original compile time.

## 5 Related Work

The Parasoft Jtest PB.LOGIC.CRRV checker is the only existing tool listed on the CERT website for the FIO08-J rule [3]. However, this commercial product was not available to us for evaluation.

FindBugs/SpotBugs [1] has a bug rule “RR: Method ignores results of `InputStream.read()`” that ensures that methods check the return value of variants of `InputStream.read()` that return the number of bytes read. The case for FIO08-J is not covered. SpotBugs also has a bug rule “INT: Bad comparison of non-negative value with negative constant or zero” that prevents the comparison of prematurely converted `chars` to the EOF value `-1`, as unsigned `chars` should not be compared to a negative value. However, neither of these rules prevents the premature conversions and the resulting defects and bad coding practices.

None of the rules in PMD, CheckStyle, and Coverity prevent the premature conversions.

## 6 Conclusions

This paper presents a qualifier-based type system that guarantees that a premature conversion from a `read int` to `byte/char` never happens at run time. We instantiated this type system for Java’s `read API`: `InputStream.read()`, `Reader.read()`, and their overrides in the JDK. We built an implementation

on top of the Checker Framework, which only required extending one type rule and three transfer functions in the framework. This implementation is available at <https://github.com/opropp/ReadChecker>. With only a very low annotation burden, this tool found 3 defects, 8 bad coding practices, and generated no false positives in 35 large, well-maintained, open source projects.

**Acknowledgements.** We thank the reviewers for their comments, which helped us to improve the paper. We also thank Daniel Caccamo, Jeff Luo, and Sadaf Tajik for feedback on drafts. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada. This material is based upon work supported by the United States Air Force under Contract No. FA8750-15-C-0010. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Air Force and the Defense Advanced Research Projects Agency (DARPA).

## References

1. Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y.: Evaluating static analysis defect warnings on production software. In: ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (2007)
2. Dietl, W., Dietzel, S., Ernst, M.D., Muslu, K., Schiller, T.W.: Building and using pluggable type-checkers. In: Software Engineering in Practice Track, International Conference on Software Engineering (ICSE), May 2011
3. Distinguish between characters or bytes read from a stream and  $-1$ . In: [4], November 2017. <https://wiki.sei.cmu.edu/confluence/display/java/FIO08-J.+Distinguish+between+characters+or+bytes+read+from+a+stream+and+-1>. Accessed 25 Nov 2017
4. Svoboda, D., Sutherland, D.F., Seacord, R.C., Mohindra, D., Long, F.: The CERT Oracle Secure Coding Standard for Java. Addison-Wesley Professional, Boston (2011)



# An Efficient Rewriting Framework for Trace Coverage of Symmetric Systems

Flavio M. De Paula<sup>(✉)</sup>, Arvind Haran, and Brad Bingham

IBM Corporation, Austin, TX, USA  
{fmdepaul,aharan,bdbingha}@us.ibm.com

**Abstract.** Verification coverage is an important metric in any hardware verification effort. Coverage models are proposed as a set of events the hardware may exhibit, intended to be possible under a test scenario. At the system level, these events each correspond to a visited state or taken transition in a transition system that represents the underlying hardware. A more sophisticated approach is to check that tests exercise specific sequences of events, corresponding to traces through the transition system. However, such trace-based coverage models are inherently expensive to consider in practice, as the number of traces is exponential in trace length. We present a novel framework that combines the approaches of conservative abstraction with rewriting to construct a concise trace-based coverage model of a class of parameterized symmetric systems. First, we leverage both symmetry and rewriting to construct abstractions that can be tailored by users' defined rewriting. Then, under this abstraction, a coverage model for a larger system can be generated from traces for a smaller system. This coverage model is of tractable size, is tractable to generate, and can be used to identify coverage-holes in large systems. Our experiments on the cache coherence protocol implementation from the multi-billion transistors IBM POWER<sup>TM</sup> Processor demonstrate the viability and effectiveness of this approach.

## 1 Introduction

Coverage analysis is a well-established verification concept on both software and hardware communities [1]. Coverage analysis evaluates how well-exercised a code is under some regression suite. This analysis can be syntactical (e.g., line-, branch-coverage) or semantical (e.g., path-coverage, functional coverage). On complex systems, the number of coverage-points is significantly large and extracting meaning of them becomes harder. For example, let's say that we have two coverage-points that were hit by a regression suite one million times and ten million times, respectively. Does that mean that the first coverage-point needs more testing? If so, how much more? It's difficult to say, but one can definitely say more tests are needed if a coverage-point was not at all hit. Thus, a large number of coverage-points can be a double-edged sword: the more coverage-points the better, but, at the same time, more difficult to extract meaning from their analysis. Some of these coverage-points may not be hittable, while others

may be redundant in some sense (e.g., one implies the other). In both cases, these strain the verification (e.g. adding unnecessary tests, simulations cycles). How can we reason about these? In particular, our main concern is with redundant coverage-points.

While a large body of research exists on coverage-metrics (e.g. [2–7]), some of which we discuss in Sect. 2, they don't necessarily address coverage redundancy (although some techniques, most notably [2,5,8], could be understood as grouping different design states that satisfy a property into equivalent classes).

To further motivate our discussion on coverage models and informally present our technique to *rewrite* redundancy, consider that a high-level specification exists and that we are tasked to write test-cases even in the absence of a fully defined specification. Assume, then, that armed with this specification we can construct an automata-based model from which we can extract *traces*. These traces compose our coverage-model of interest. Now, as an example, consider a system composed of three symmetric sub-systems communicating through a channel. These systems exchange messages among each other using some arbitrary protocol allowing for request, acknowledge and back-off signals to control the dataflow among them. Let's assume each of these systems contain a state-machine and a buffer to hold the transmitted/received messages. When one buffer is about to get full, a back-off signal is sent to the transmitting system to prevent overflow. Exchanged messages can have variable length. Assume there can only be one transmitter at a time. The other two sub-systems may be either both receiving messages or one receiving and one in idle state.

One coverage model with emphasis on the function of the overall system would account for every possible: permutations of transmitter, receiver and idle sub-systems' state; permutation of controlling signals; and, messages including different sizes. Although the proposed coverage model captures every possible behavior of the system, its level of detail may lead to redundancy. For instance, at some point during the design it might be sufficient to simply know that the system properly processes certain types of messages regardless of its size. In other words, *types of messages* become one or more equivalence class (e.g., message-transmitted, message-received-with-error) of a possibly larger set of coverage-points. Likewise, recall that the system is symmetric, thus capturing every possible state permutation as a coverage-point may also lead to redundancy. We address the former case by means of *rewriting*. The user can define rewrite-rules that effectively construct classes of equivalent-behaviors. Each class may represent a large set of individual coverage-points on the original design. We address the latter by means of a novel abstraction technique which leverages symmetry and other properties found in common communicating systems (e.g., client-server, cache-protocols) under some special restrictions (described in Sect. 5).

In our experience, combining these two key ideas, rewriting and the novel abstraction proved to be very efficient in constructing a concise coverage model and helpful in finding coverage-holes during the verification of the IBM POWER<sup>TM</sup> Processor.

The main contribution of this paper is a novel approach for trace coverage of symmetric systems, that

- allows a small system to be used to generate a coverage model for any larger model;
- allows user-defined rewriting, in order to tailor the technique;
- is successfully applied to the verification of the cache-coherence protocol of an IBM POWER<sup>TM</sup> Processor;
- is extendable to several other cache-coherence protocols.

We note that this work is motivated by our case study, cache coherence at the system level with highly abstracted states. While this work is directly applicable to coverage for other cache protocols at this level, we note that the general framework is applicable to very different problems, and may be customized according to what interesting coverage means. This allows for users to define their own abstraction approaches and utilize the framework, provided they fill in the theory that supports Theorem 1, which establishes that traces from a small system can be used as a coverage model for a larger system.

This paper is organized as follows: next we present work related to coverage and symmetry reduction techniques. Section 3 gives the foundations of our formalism and introduces a running-example based on the MESI cache-coherence protocol [9]. Section 4 gives the high-level view of the approach. Section 5 contains the technical details of the abstraction and the main theoretical result. After these, we present our experiments as applied to the IBM POWER<sup>TM</sup> Processor, followed by conclusions.

## 2 Related Work

In the context of automata-based coverage, we can broadly group approaches with similar goals. Some research focus on property coverage, i.e., Chockler et al. [5], and Dwyer et al. [2]. Regardless of the chosen formalism, these work share with ours the intent of covering an abstraction of the actual system, that is, in their case, each property defines a class of equivalent behaviors. These approaches, while more general in scope, are more susceptible to scalability issues (as other formal verification frameworks). In contrast, because our approach focuses on specific types of systems, we can take advantage of properties like symmetry to reason about very large systems.

The work presented by Shen and Abraham [10] talks about defining coverage models comprising of finite sequences of state transitions of an FSM extracted from the RTL of the design. While both works use the notion of bounded-length trace coverage, our approach operates on the protocol-level and uses a specification of the design to generate the coverage models. We then use a large system-level simulation framework to measure trace coverage. In addition, our key contribution is an abstraction technique that reduces the number of coverage entries by utilizing properties of a class of symmetric systems. This allows the practical use of trace coverage in large scale designs.

Finally, in regards to work related to symmetric systems, the approach presented by Chou et al. [11] for parameterized symmetric systems bears some similarities to our approach, in that we consider abstraction of such systems. Their state abstraction keeps a fixed number of agents modeled exactly, with transitions overapproximated due to hidden “others”. In contrast, our state abstraction approach is a “folding” that maintains only that one-or-more agent are in each particular state. Their goal is to formally verify that a symmetric system satisfies a parameterized state predicate by way of human-guided refinement, whereas we aim to reason about trace coverage.

### 3 Preliminaries

We use the well-established MESI cache-coherence protocol as a running-example to aid in the presentation of the more complex concepts introduced in this paper. It considers cachelines to be shared (S) among different caches; to be exclusively present in one cache, being modified (M) or not (E); or to be invalid (I). A *transition-system*, defined next, appears in of Fig. 1a and illustrates the possible state transitions for the MESI protocol.

**Definition 1.** A *transition-system* is a tuple  $(\mathcal{S}, \text{init}, \Sigma, \mathcal{R})$  where

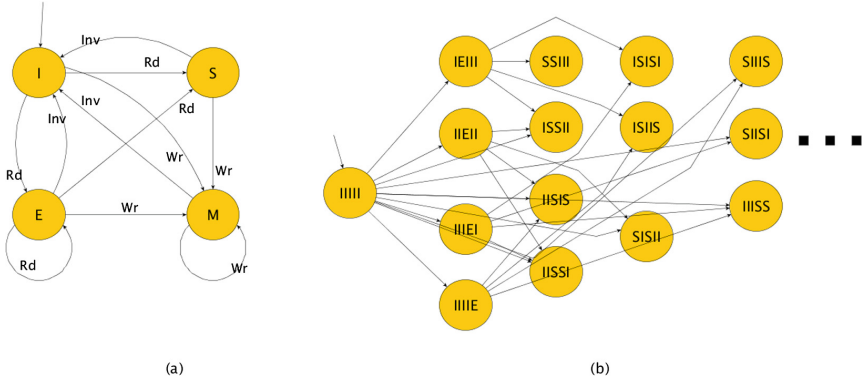
- $\mathcal{S}$  is the set of states,
- $\text{init} \in \mathcal{S}$  is the initial state,
- $\Sigma$  is the set of symbols,
- $\mathcal{R} : (\mathcal{S} \times \Sigma) \rightarrow \mathcal{S}$  is the (partial) transition function.

We model the *composed-system* (CS) as symmetric copies of a *transition-system*. A CS depends on a transition-system, the number of symmetric transition-systems  $n$  (i.e., the *size*), and a partition of symbols into actions and reactions.

**Definition 2.** A *composed-system* is a tuple  $(\mathcal{S}, \mathbf{q}_0, \mathbb{A}, \mathbb{R})$  where:

- $\mathcal{S} = \mathcal{S}^n$  is the set of composed-states (or c-states),
- $\mathbf{q}_0 = (\text{init}, \dots, \text{init})$  is an  $n$ -tuple and the initial c-state,
- $\mathbb{A} \subseteq \Sigma^n$  is the set of composed-symbols (see below),
- $\mathbb{R} : (\mathcal{S} \times \mathbb{A}) \rightarrow \mathcal{S}$  is the composed transition function (see below).

For both c-state and composed-symbols, we use subscripting to refer to a particular component, i.e.  $s_i$  is component  $i$  of c-state  $\mathbf{s}$ . Composed-symbols are restricted such that one system is the *actor* while all others are *reactors*, according to  $r : \Sigma_{\text{act}} \rightarrow \Sigma_{\text{react}}$ , where  $\Sigma_{\text{act}}$  and  $\Sigma_{\text{react}}$  are a partition of  $\Sigma$ . That is, all elements of  $\mathbb{A}$  are some permutation of  $(c, r(c), \dots, r(c))$  for some  $c \in \Sigma_{\text{act}}$ . Transition function  $\mathcal{R}$  must have a defined mapping for elements  $\mathcal{S} \times \Sigma_{\text{react}}$ , but may be undefined for elements of  $\mathcal{S} \times \Sigma_{\text{act}}$ . In Fig. 1b, we have a fragment of the CS of size 5 (the entire CS is too large to be shown). The composed transitions are written using in-fix notation; where  $\mathbf{s} \xrightarrow{\mathbf{a}} \mathbf{s}'$  means that  $\mathbb{R}(\mathbf{s}, \mathbf{a}) = \mathbf{s}'$ . Composed transitions are component-wise application of  $\mathcal{R}$ :



**Fig. 1.** (a) MESI cache-protocol; (b) subgraph of the CS of size 5. Edge labels and self-loop edges are not represented.

$$\mathbf{s} \xrightarrow{\mathbf{a}} \mathbf{s}' \Leftrightarrow \forall i. \mathcal{R}(s_i, a_i) = s'_i.$$

This leads to a natural definition of a CS trace.

**Definition 3.** A trace of CS  $\mathcal{M}$  is a finite, alternating string  $\mathbf{s}_0 \mathbf{a}_0 \mathbf{s}_1 \mathbf{a}_1 \dots \mathbf{a}_{\ell-1} \mathbf{s}_\ell$  of c-states and composed-symbols such that

- $\mathbf{s}_0 = \mathbf{q}_0$ ;
- $\mathbf{s}_i \xrightarrow{\mathbf{a}_i} \mathbf{s}_{i+1}$  for  $0 \leq i < \ell$ .

Let  $\text{TRACES}(\mathcal{M})$  denote the set of traces of CS  $\mathcal{M}$ . The length of trace  $w$ , denoted with  $|w|$ , is the number of composed-symbols that appear in it (with repetition). The *tail* of a trace  $tl(w)$  is the last character appearing in  $w$ , necessarily a state; the *head* of a trace  $hd(w)$  is the first character appearing in  $w$ , necessarily a state. A *subtrace* is a substring of a trace in  $\text{TRACES}(\mathcal{M})$  beginning and ending in a state. A c-state is *reachable* if it appears in some trace of  $\text{TRACES}(\mathcal{M})$ .

**Definition 4.** Given set  $A$  and positive integer  $n$ , a permutation  $\pi$  is a bijection  $A^n \rightarrow A^n$  that reorders the components according to one-to-one function  $\xi : \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, n\}$ , where

$$\pi((a_1, a_2, \dots, a_n)) = (a_{\xi(1)}, a_{\xi(2)}, \dots, a_{\xi(n)}).$$

Definition 4 allows us to consider permutations over both elements of  $\mathbb{S}$  and elements of  $\mathbb{A}$ . We extend the notion of permutation to traces: if trace  $w = \mathbf{s}_0 \mathbf{a}_0 \dots \mathbf{a}_{\ell-1} \mathbf{s}_\ell$ , then  $\pi(w) = \pi(\mathbf{s}_0) \pi(\mathbf{a}_0) \dots \pi(\mathbf{a}_{\ell-1}) \pi(\mathbf{s}_\ell)$ . A set of traces  $T$  is *closed under permutation* when for all permutations  $\pi$ ,  $w \in T$  if and only if  $\pi(w) \in T$ . The symmetry of  $\mathcal{M}$  leads to an equivalence on traces modulo permutation. Figure 2a depicts a CS for MESI modulo permutation. Note that each state in this figure is a representative of a possible permutation (e.g., state SIII represents itself, ISIII, IISII, IIIIS and IIIIS from Fig. 1b).

**Proposition 1.** *Given CS  $\mathcal{M}$  the set  $\text{TRACES}(\mathcal{M})$  is closed under permutation. Likewise, the subtraces of  $\mathcal{M}$  are closed under permutation.*

Now that we have defined CSs and traces, we can then present our strategy for efficiently computing trace coverage.

## 4 Strategy Overview

Our goal is to reason about CS traces in a comprehensive and yet concise way. From practice we leverage two key insights: that regression tests are finite in length and that many system traces share some common behavior. Then, we compute canonical representatives for a set of traces, called an abstract trace. The c-states are abstracted to maintain the distinct states in their composition, but may obfuscate the count of each state; it follows that traces that are equal up to permutation have the same representative (e.g., blue ellipses in Fig. 2b encompassing SSIII, SSSII and SSSSI). Our presentation focus on CSs adhering to specific rules, namely that at most two states may occur more than once in a reachable c-state. For MESI, these states are S and I (note in Fig. 2b the “squared” states side-by-side with the groupings they represent). These rules are motivated by our experience with real cache protocols, and to simplify the presentation<sup>1</sup>. The key novelty is how we combine the approaches of rewriting and abstraction into a single framework to empower users to easily define their coverage models.

### 4.1 Rewriting Systems

We leverage the concept of rewriting for computing canonical coverage-points. Informally, a rewriting system is a set and a list of transformations on set elements. Rewriting works by the successive applications of transformations to objects until they are no longer applicable.

The class of rewriting systems we are interested in are both *terminating* and *confluent*, necessary and sufficient conditions for finding canonical representatives. In our case, the set comprises of traces and the transformations are string-replacement rules.

**Definition 5.** *A Rewriting System (RS) is  $(\Gamma^*, \rho)$ :*

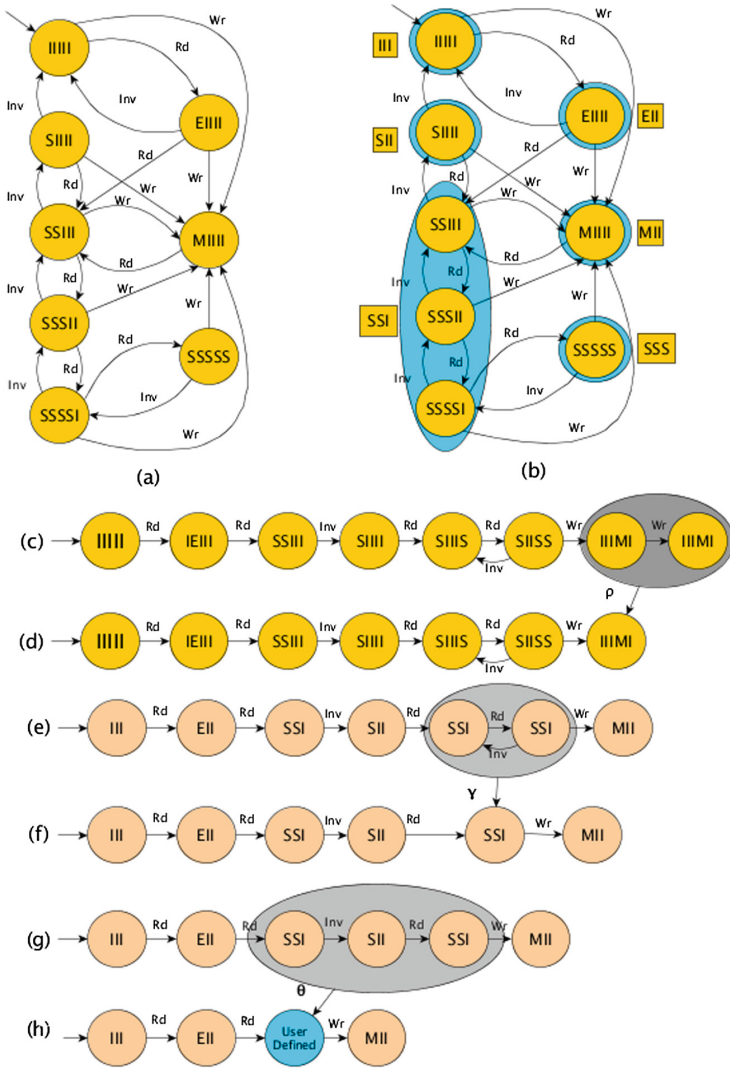
- $\Gamma$  is a finite alphabet,
- $\rho \subseteq \Gamma^* \times \Gamma^*$  is a relation on strings.

Each element  $(l, r) \in \rho$  is called a rewrite rule, applied as follows: for any  $u, v \in \Gamma^*$ ,  $u \rightarrow v$  iff there exists  $(l, r) \in \rho$  such that for some  $x, y \in \Gamma^*$ ,  $u = xly$  and  $v = xry$ . The notation  $\xrightarrow{*}$  is the reflexive and transitive closure of  $\rightarrow$ .

---

<sup>1</sup> Other systems that focus on different abstractions require their own tailored restrictions and proof of Theorem 1.





**Fig. 2.** Running example illustration. Vertices in (a) and (b) are sets of c-states that are equal up to permutation; vertices in (c) and (d) are c-states; vertices in (e) through (g) are abstract states; vertex “User Defined” is a representative of some abstract states. Edges of (a) through (d) are labeled not with composed-symbols, but the unique action component. (a) A representation of the MESI CS of size 5 from Fig. 1b, (self-loops omitted). (b) Equivalence classes created by state abstraction; abstract state SSI is a member of *Blur*. (c) A representation of a set of CS traces, where each trace traverses the cycle on SIIIS and SIISS a different number of times. (d) A set of CS traces after stutter removal. (e) The corresponding abstract trace after state abstraction. (f) The corresponding abstract trace after abstract trace rewriting. (g, h) Abstract traces before and after optional user-defined rewriting that combines a sequence of abstract states where S-state and I-state toggling occur.

A RS is *terminating* if there is no infinite sequence  $x_0, x_1, \dots$  such that for all  $i \geq 0$ ,  $x_i \rightarrow x_{i+1}$ . Termination can be established by finding an ordering function (e.g., string length) that all rewrite rules obey. RS  $(\Gamma, \rho)$  is *confluent* if for all  $w, x, y \in \Gamma^*$ , the existence of reductions  $w \xrightarrow{*} x$  and  $w \xrightarrow{*} y$  implies there exists a  $z \in \Gamma^*$  such that  $x \xrightarrow{*} z$  and  $y \xrightarrow{*} z$ .

Rewriting theory tells us that in a terminating and confluent RS (a *TCRS*), any string can be reduced to a unique normal form by applying rewrite rules until the object is irreducible [12, 13].

For elements of  $\text{TRACES}(\mathcal{M})$ , we define a TCRS that removes stuttering actions. Here, the alphabet  $\Gamma$  is  $\mathbb{S} \cup \mathbb{A}$ , and the relation  $\rho$  is as follows.

$$\forall \mathbf{s} \in \mathbb{S}. \forall \mathbf{a} \in \mathbb{A}. (\mathbf{sas}, \mathbf{s}) \in \rho \quad (1)$$

We encapsulate this TCRS as function  $\tau_n$ , which is used to map traces  $w$  of  $\mathcal{M}$  with size  $n$  to a representative trace  $\tau_n(w)$  without stuttering. In Fig. 2c, the last two states in this trace stutter and, thus, can be rewritten by applying Eq. 1 to arrive at Fig. 2d.

## 4.2 Abstraction and Coverage Model

We leverage insights from symmetric systems and rewriting into a new abstraction, composed of three steps: trace rewriting, then folding, followed by a second trace rewriting. The added flexibility to users to define their coverage models comes from the ability to apply rewriting on the resulting abstract traces.

Given a CS, consider *Ref* the CS with fixed size  $k$  and *DUT*, the CS for some fixed size  $n > k$ . We will establish the property  $w \in \text{TRACES}(\text{Ref})$  implies that there is a trace  $w' \in \text{TRACES}(\text{DUT})$  with the same abstract representative. This allows coverage-holes to be identified in any test subset of  $\text{TRACES}(\text{DUT})$  by comparing their abstract representatives with those of  $\text{TRACES}(\text{Ref})$ .

## 5 Conservative Abstraction

Here we will define  $\alpha$ , an abstraction function on traces of  $\mathcal{M}$ , and establish the main result.

### 5.1 Abstractable Composed-Systems

In this Section, we define an abstraction on traces that applies to the CSs that adhere to specific rules. A CS that complies with these rules is *abstractable*. In essence, reachable c-states are restricted to only have two states that appear more than once.

Partition  $\mathcal{S}$  into two sets,  $\mathbf{X} = \{X_1, X_2, \dots, X_m\}$  for  $m \geq 1$  and  $\mathbf{Y} = \{Y_1, Y_2\}$ . Associate with each state a *ranking*, where the rank ordering  $\succ$  is

$$X_1 \succ X_2 \succ \dots \succ X_m \succ Y_1 \succ Y_2.$$

Noting that each transition has a distinct *actor* and the others are *reactors*, the following characterizes CSs that may be abstracted.

**Definition 6.** A CS is called *abstractable* if the initial state is  $n$ -tuple  $(Y_2, \dots, Y_2)$  and the transitions adhere to the following rules.

1. Reactors do not increase rank.
2. If actor  $Y_1$  decreases rank to  $Y_2$ , reactors do not change.
3. If any actor increases rank to some  $\mathbf{X}$ -state, reactors must change to  $Y_2$ .
4. If actor  $Y_2$  increases rank to  $Y_1$ , reactors  $X_i$  for  $1 \leq i < m$  decrease rank; reactor  $X_m$  does not change.

**Lemma 1.** If a CS is abstractable, then any reachable  $c$ -state

1. includes at most one element of  $\mathbf{X}$ , and
2. if it includes  $X_i$  for  $i \neq m$ , it includes at most  $i - 1$  states that are  $Y_1$ .

*Proof (Sketch).* Initial transition to a  $c$ -state including  $X_i$  has no  $Y_1$  states; subsequent transitions increase  $i$  by at least 1 and the number of  $Y_1$  increases by at most 1.

From the MESI running-example:  $\mathbf{X} = \{M, E\}$  and  $\mathbf{Y} = \{S, I\}$ . That is,  $m = 2$ ,  $X_1 = M$ ,  $X_2 = E$ ,  $Y_1 = S$  and  $Y_2 = I$ . Their ranking naturally follows:  $M \succ E \succ S \succ I$ . Therefore, Lemma 1 is saying that an abstractable state can contain no more than one M or one E. Second, when M is present and since  $m = 2$ , there cannot be any S. The last vertex in Fig. 2e shows such abstraction from Fig. 2d.

## 5.2 Trace Abstraction

We define the abstract states as strings of length  $k = m + 1$  where each symbol is a state, states are listed in nonincreasing rank order, and the corresponding  $c$ -state is reachable according to Lemma 1. That is,  $\mathbb{S}^A$  is all strings of the form  $cY_1^aY_2^b$  where  $c$  is any state of  $\mathcal{S}$ , with  $c = X_i$  with  $i \neq m$  implies that  $a < i - 1$  and  $c = Y_2$  implies that  $a = 0$ . All permutations of a  $c$ -state map to the same abstract state, so we only consider a specific permutation without loss of generality (see Fig. 2e).

**Definition 7 (State Abstraction).** For fixed  $n \geq k$ , let  $\phi_n$  be a function from reachable  $c$ -states of  $\mathbb{S}$  to  $\mathbb{S}^A$ . Let  $\mathbf{s} = (s_1, \dots, s_n)$  be a  $c$ -state with states in nonincreasing rank order. Then  $\phi_n(\mathbf{s})$  is the unique element of  $\mathbb{S}^A$  according to

$$\phi_n(\mathbf{s}) = \begin{cases} s_1s_2 \dots s_k, & \text{when all } s_{k+1}, \dots, s_n \text{ are equal;} \\ s_1s_2 \dots s_{k-1}Y_2, & \text{otherwise.} \end{cases}$$

By construction, some elements of  $\mathbb{S}^A$  are “exact” in that they have a single  $c$ -state in its preimage, up to permutation, for each  $n$ . We call this set *Exact*, and define the remaining elements *Blur* as  $\mathbb{S}^A \setminus \text{Exact}$ . The string characterization of *Blur* follows directly from Lemma 1 and Definition 7.

**Proposition 2.** The set *Blur* are the abstract states with exactly one  $Y_2$ , i.e.,  $\{Y_1^{k-1}Y_2, X_mY_1^{k-2}Y_2\}$ .

For example, the trace in Fig. 2-e has blur states SSI and SII. Next, we define the abstract symbols  $\mathbb{A}^A$  as the set of actor symbols  $\Sigma_{\text{act}}$ . Associated symbol abstraction function maps the element of  $\mathbb{A}$  to its component that is an action symbol.

**Definition 8** (*Symbol Abstraction*). For fixed  $n \geq k$ , let  $\psi_n : \mathbb{A} \rightarrow \mathbb{A}^A$  map composed-symbol  $\mathbf{a}$  to the unique element  $a_i \in \Sigma_{\text{act}}$ .

With Definitions 7 and 8 in place, we define function  $\delta_n$  that maps traces to abstract traces.

**Definition 9** (*Trace Abstraction*). Given CS  $\mathcal{M}$  and  $w = \mathbf{s}_0 \mathbf{a}_0 \mathbf{s}_1 \mathbf{a}_1 \dots \mathbf{a}_{\ell-1} \mathbf{s}_\ell \in \text{TRACES}(\mathcal{M})$ , then  $\delta_n(w)$  is the alternating sequence

$$\phi_n(\mathbf{s}_0) \psi_n(\mathbf{a}_0) \phi_n(\mathbf{s}_1) \psi_n(\mathbf{a}_1) \dots \psi_n(\mathbf{a}_{\ell-1}) \phi_n(\mathbf{s}_\ell).$$

### 5.3 Abstract Trace Rewriting

Rewriting of the abstract traces is necessary to obfuscate the size of the CS, as actor transitions between  $Y_1$  and  $Y_2$  may have no reaction, and appear as stuttering in the abstract trace (back-to-back SSI states in Fig. 2e). Such stuttering substraces must have length that is at least the size of the CS. This is troublesome for coverage when our goal is to compare abstract traces from CSs with different size. A TCRS  $(\Gamma, \rho)$  has  $\Gamma = \mathbb{S}^A \cup \mathbb{A}^A$ , and

$$\forall s \in \text{Blur}. \forall a \in \mathbb{A}^A. (sas, s) \in \rho. \quad (2)$$

We encapsulate this rewriting system as function  $\gamma$ , which is used to map abstract traces to their unique representative abstract trace, by repeated application of the rules of  $\rho$ .

### 5.4 Coverage Holes

The trace representative function  $\alpha$  is the composition of functions<sup>2</sup>  $\gamma \circ \delta_n \circ \tau_n$ ; we forgo the  $n$  subscript as the size of  $\mathcal{M}$  will be clear from context. We overload  $\alpha$  to apply directly to CSs, where  $\alpha(\mathcal{M})$  is  $\{\alpha(w) \mid w \in \text{TRACES}(\mathcal{M})\}$ . The following Lemma relates a transition of *Ref* with a corresponding subtrace of *DUT* under the  $\alpha$  mapping. We consider the specific case where a subtrace  $\mathbf{sas}'$  of *Ref* is not affected by either rewriting function. That is,

$$\begin{aligned} \alpha(\mathbf{sas}') &= \gamma(\delta_k(\tau_k(\mathbf{sas}')))) = \gamma(\delta_k(\mathbf{sas}')) \\ &= \gamma(\phi_k(\mathbf{s})\psi_k(\mathbf{a})\phi_k(\mathbf{s}')) = \phi_k(\mathbf{s})\psi_k(\mathbf{a})\phi_k(\mathbf{s}') \end{aligned}$$

**Lemma 2.** Let  $\mathbf{s} \xrightarrow{\mathbf{a}} \mathbf{s}'$  be a transition of *Ref* with  $\alpha(\mathbf{sas}') = \phi_k(\mathbf{s})\psi_k(\mathbf{a})\phi_k(\mathbf{s}')$ . For each  $c$ -state  $\mathbf{r}$  of *DUT* where  $\phi_n(\mathbf{r}) = \phi_k(\mathbf{s})$ , there exists subtrace  $w_r$  such that  $\alpha(w_r) = \alpha(\mathbf{sas}')$ .

<sup>2</sup> Recall,  $\tau_n$  is trace rewriting;  $\delta_n$  is trace abstraction;  $\gamma$  is abstract trace rewriting.

*Proof (Sketch).* By Proposition 1, assume without loss of generality that  $\mathbf{s}$  and  $\mathbf{r}$  both have their states listed in nonincreasing order. **Case 1:** If  $\phi_k(\mathbf{s}) \in Exact$ , then  $\mathbf{r}$  is unique up to permutation. The claim holds by taking analogous action to  $\mathbf{a}$  on  $\mathbf{r}$ . **Case 2:** If  $\phi_k(\mathbf{s}) \in Blur$ , the abstract state necessarily changed, as abstract rewriting did not apply. If the actor transitions from or to some  $X_i$ , then an analogous action suffices on  $\mathbf{r}$ . Otherwise, there is a series of actions that transition a  $Y_1$  to  $Y_2$ , or a  $Y_2$  to  $Y_1$ . The resulting subtrace  $w_r$  has all states abstract to  $\phi_k(\mathbf{s})$ , except  $tl(w_r)$ , which abstracts to  $\phi_k(\mathbf{s}')$ .

Using Lemma 2, we prove the main result of this Section.

**Theorem 1.** *For each  $w \in \text{TRACES}(Ref)$ , there exists  $w' \in \text{TRACES}(DUT)$  such that  $\alpha(w) = \alpha(w')$ .*

*Proof.* By induction on the length of  $w$ . If  $|w| = 0$ , then  $w = \mathbf{q}_0$ ;  $\alpha(\mathbf{q}_0) = Y_2^k$ . The initial c-state  $\mathbf{q}_0$  of  $DUT$  is  $(Y_2, Y_2, \dots, Y_2)$ , and  $\alpha(\mathbf{q}_0) = Y_2^k$ . Therefore,  $w' = \mathbf{q}_0$  satisfies the claim. For  $|w| > 0$ , assume the claim holds for all  $w$  with length  $\ell$ . We consider arbitrary trace  $w$  of  $Ref$  of length  $\ell + 1$ . Note that  $w = \tilde{w}\mathbf{a}_\ell\mathbf{s}_{\ell+1}$ , where the claim holds for length  $\ell$  trace  $\tilde{w}$ , and  $tl(\tilde{w}) = \mathbf{s}_\ell$ , and the transition  $\mathbf{s}_\ell \xrightarrow{\mathbf{a}_\ell} \mathbf{s}_{\ell+1}$  necessarily exists. We consider 3 cases on the subtrace  $\mathbf{s}_\ell\mathbf{a}_\ell\mathbf{s}_{\ell+1}$ . **Case 1:** If  $\mathbf{s}_\ell = \mathbf{s}_{\ell+1}$ , trace rewriting will map the subtrace  $\mathbf{s}_\ell\mathbf{a}_\ell\mathbf{s}_{\ell+1}$  to  $\mathbf{s}_\ell$ , i.e. the rewritings of  $\tilde{w}$  and  $w$  are equal. Therefore  $\alpha(\tilde{w}) = \alpha(w)$  and the claim holds. **Case 2:** Else if  $\phi_k(\mathbf{s}_\ell) \in Blur$  and  $\phi_k(\mathbf{s}_\ell) = \phi_k(\mathbf{s}_{\ell+1})$ , abstract trace rewriting will map the substring  $\phi_k(\mathbf{s}_\ell)\psi_k(\mathbf{a}_\ell)\phi_k(\mathbf{s}_{\ell+1})$  to  $\phi_k(\mathbf{s}_\ell)$ , i.e. the abstract rewritings of  $\delta_k(w)$  and  $\delta_k(\tilde{w})$  are equal. Therefore  $\alpha(\tilde{w}) = \alpha(w)$  and the claim holds. **Case 3:** Else, neither of the rewritings apply to the subtrace, and  $\alpha(\mathbf{s}_\ell\mathbf{a}_\ell\mathbf{s}_{\ell+1}) = \phi_k(\mathbf{s}_\ell)\psi_k(\mathbf{a}_\ell)\phi_k(\mathbf{s}_{\ell+1})$ . Let  $\tilde{w}'$  be the  $DUT$  trace such that  $\alpha(w') = \alpha(\tilde{w}')$ . By Lemma 2, for each  $\mathbf{r} \in \phi_n^{-1}(\phi_k(\mathbf{s}_\ell))$  there exists subtrace  $w_r$  with  $\alpha(w_r) = \alpha(\mathbf{s}_\ell\mathbf{a}_\ell\mathbf{s}_{\ell+1})$  and  $hd(w_r) = \mathbf{r}$ . Choosing  $\mathbf{r}$  as  $tl(\tilde{w}')$ , compose  $\tilde{w}'$  with  $w_r$  to create requisite trace  $w'$ .

**Corollary 1.** *Let function  $\theta$  be associated with TCRS on abstract traces. For each  $w \in \text{TRACES}(Ref)$ , there exists  $w' \in \text{TRACES}(DUT)$  such that  $\theta(\alpha(w)) = \theta(\alpha(w'))$ .*

For example, Fig. 2h represents a user-defined rewriting rule  $\theta$  applied to Fig. 2g. Here, the “user” is succinctly capturing that at least one subsystem is toggling between two abstract states.

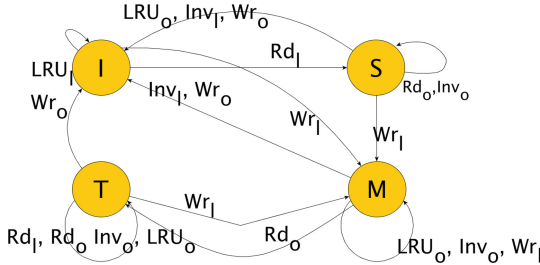
## 6 Experiments

It follows from Theorem 1 that it suffices to reason about coverage-holes of a potentially large  $DUT$  in terms of a much smaller  $\alpha(Ref)$ . In order to support this claim, we present the results from a case study where we demonstrate the applicability of our abstraction technique on an IBM POWER™ Processor. In Sect. 6.1, we outline the MIST protocol transition-system for cache coherence.

Section 6.2, then showcases the viability of using  $\alpha$  in creating an abstracted coverage model that succinctly represents concrete traces of a CS based on MIST. We then conclude by illustrating the effectiveness of this coverage model for the verification of cache-coherence of an IBM POWER<sup>TM</sup> Processor.

### 6.1 The MIST Protocol

The MIST protocol is a simplified version of an IBM POWER<sup>TM</sup> Processor cache-coherence protocol [14, 15] that is based on the well-known MOESI protocol [16]. This implementation has four states: **M**, **I**, **S** and **T**<sup>3</sup>, shown in Fig. 3.



**Fig. 3.** Simplified IBM POWER<sup>TM</sup> Processor Cache-Coherence Protocol with 4 states: Modified, Invalid, Shared and Tagged. Operations: **Inv** (invalidate), **Rd** (read), **Wr** (write), **LRU**; Mode: **l** (local), **o** (other). Missing edges are self-loops.

Formally, using notation from Sect. 3, the MIST protocol has the following transition-system

- $\mathcal{S} = \{\mathbf{M}, \mathbf{I}, \mathbf{S}, \mathbf{T}\}$ ,
- $init = \mathbf{I}$ ,
- $\Sigma = \{\mathbf{noop}_x, \mathbf{Inv}_x, \mathbf{Rd}_x, \mathbf{Wr}_x, \mathbf{LRU}_x\}$ , where  $x \in \{\mathbf{l}, \mathbf{o}\}$ . Subscripts **l** and **o** indicate the actor and reactor type of the action, respectively. Assume  $\Sigma_{act}$  is the set of ‘actor’ type actions (subscript **l**). Note that for our experiments (in Sects. 6.3, 6.4, 6.5) we choose a subset of actions of the IBM POWER<sup>TM</sup> Processor cache coherence protocol where  $|\Sigma_{act}| = 15$ . In the interest of IP non-disclosure we obscure these particular elements of  $\Sigma$ . However, the elements of  $\Sigma$  can be mapped to the set  $\{\mathbf{Inv}, \mathbf{Rd}, \mathbf{Wr}, \mathbf{LRU}, \mathbf{noop}\}$ .
- $\mathcal{R}$ : See Fig. 3.

The CS for the MIST protocol is parameterized by  $n \geq 3$ , and constructed with the preceding transition system and reaction  $r : r(\mathbf{Act}_l) = \mathbf{Act}_o$  where  $\mathbf{Act} \in \{\mathbf{Inv}, \mathbf{Rd}, \mathbf{Wr}, \mathbf{LRU}, \mathbf{noop}\}$ . The MIST system is *abstractable* with the

<sup>3</sup> A cacheline is in state **M** when it has been modified; **I** when invalidated; **S** when shared; and, **T** when the dirty cacheline is possibly being shared with other nodes while *this* owner is responsible for servicing requests for sharing the cacheline.

partition of  $\mathcal{S}$  into two sets  $\mathbf{X} = \{\mathbf{M}, \mathbf{T}\}$  (where  $X_1 = \mathbf{M}$ ,  $X_2 = \mathbf{T}$ ) and  $\mathbf{Y} = \{\mathbf{S}, \mathbf{I}\}$  (where  $Y_1 = \mathbf{S}$ ,  $Y_2 = \mathbf{I}$ ) (as defined in Sect. 5.1). For any CS based on MIST with  $n \geq 3$ , henceforth  $\mathcal{M}$ , the state abstraction function,  $\phi_n$  (from Definition 7) defines the set of abstract states  $\{\mathbf{III}, \mathbf{SII}, \mathbf{SSI}, \mathbf{SSS}, \mathbf{MII}, \mathbf{TSI}, \mathbf{TSS}\}$ .

Cache-coherence verification of IBM POWER<sup>TM</sup> Processor is incomplete without stimulating the system to transition between c-states with a single owner to c-states with multiple shared agents under different types of actions, as these behaviors cover state spaces of the protocol that may lead to coherency issues. Our abstraction enables us to define coverage models that help track these properties in a system of arbitrary size, but with a smaller model. Note that the abstraction  $\alpha$  can also be extended to commonly known cache protocols (such as MOESI, MESI etc.) with appropriate partitions of  $\mathcal{S}$ .

## 6.2 Model Viability

In order for a coverage model to be usable for IBM POWER<sup>TM</sup> Processor, the number of coverage-points (henceforth, *size* of coverage model) and the time taken to generate them needs to be tractable. In Table 1, we compare the coverage model sizes, with and without our abstraction ( $\alpha$ ), for traces based on a MIST experimental setup. To illustrate the complexity of trace coverage, we also showcase the theoretical maximum size of a coverage model comprising of concrete traces of a CS based on a MIST experimental setup (which is  $|\mathcal{S}^n| \times (n \times |\Sigma_{\text{act}}|)^\ell$ , for trace-length  $\ell$  and allowing for initialization to arbitrary reachable states in  $\mathcal{S}^n$ ). Note that this theoretical size is much larger than the other coverage models, since it allows for non-permissible transitions of a practical MIST-based system (as defined in Sect. 6.1). We also only show the coverage model sizes that do not include the **noop** transitions, which is the setup we use for our case study. The coverage model definitions and abstraction techniques were prototyped/implemented using custom software (written in C++, Python) that is compatible with the IBM POWER<sup>TM</sup> Processor verification infrastructure. The experiments were conducted on a single core *Intel Xeon E312xx (Sandy Bridge)* (CPU MHz: 2899.9 Cache size: 4096 KB Memory: 106.89 GB). As we observe, the proposed abstraction significantly reduces the coverage model size with a tolerable overhead to the setup time for the coverage model. As always, further optimization can be applied to improve model memory footprint and runtime which is beyond the scope of this paper.

## 6.3 Effectiveness

To show that the presented technique is effective in the use for analyzing coverage in large scale designs, we conducted two experiments involving a version of a cache-coherence monitor (henceforth, *the monitor* [17]) used in IBM POWER<sup>TM</sup> Processor verification. The monitor is part of the verification framework for IBM POWER<sup>TM</sup> Processor and is used to perform certain (micro)architectural checks related to cache coherence. The monitor observes RTL simulation (by means of *simulation logs*) and then produces a *pass* or a *fail* result (w.r.t. its checks). The

**Table 1.** Comparing the number of coverage-points of different MIST coverage models and time taken (in hours) to generate the coverage models. For all cases,  $|\mathcal{S}|$  is 4 and trace length  $\ell = 4$ . N/A reflects the fact that  $\alpha$  for MIST requires  $n \geq 3$ .

	n = 2		n = 3		n = 4	
	No.	Time	No.	Time	No.	Time
Theory	$\approx 10^8$	N/A	$\approx 10^9$	N/A	$\approx 10^{10}$	N/A
No $\alpha$	959,207	0.03	9,355,651	0.59	60,104,523	5.15
With $\alpha$	N/A	N/A	270,814	0.86	296,270	7.16

monitor also produces, for each cache-line exercised in the simulation, a *witness* trace  $w \in \text{TRACES}(\mathcal{M})$ . This witness trace is a representation of the behavior monitored/checked on the cache-line. The monitor can be said to vacuously pass if it produces a pass result but the witness trace (or its abstraction) is not representative of the behavior simulated. The monitor is said to have *covered* an abstract trace  $t \in \alpha(\text{TRACES}(\mathcal{M}))$  if there exists an input stimulus for which the monitor produced a witness such that  $\alpha(w) = t$ .

In order to avoid missing bugs in RTL simulation, it is important to verify the implementation of the monitor under different behaviors over a cacheline. Particularly, vacuous passes indicate bugs in the implementation of the monitor that may mask some real hardware issues. To this end, the first experiment, Sect. 6.4, evaluates the monitor with respect to a reference model (*Ref*) in an unit-testing environment. That is, we want to know whether this monitor *covers* all the behaviors prescribed by *Ref* and whether coverage-holes are indeed insightful. The second experiment, Sect. 6.5, places the monitor in a test-bench used for the IBM POWER<sup>TM</sup> Processor and allows us to analyze the monitors trace-coverage on inputs produced by the *DUT* with respect to *Ref*. This setup allows us to measure what is being checked in simulation using the monitor as well as what is being stimulated by the test cases. We discuss some insights from the results.

## 6.4 Monitor-Centric Experiments

To demonstrate applicability of our technique to evaluate the monitors coverage of operations on a cacheline, we start with a “ $3 \times 3$ ” reference model *Ref* ( $n = 3$  and  $\ell = 3$ ). The reference model only allows valid behaviors (validity as defined by the specification). We exhaustively generated the set,  $\text{TRACES}(\text{Ref})$ , up to length  $\ell = 3$  ( $|\text{TRACES}(\text{Ref})| = 990,648$ ). We use the set  $\alpha(\text{TRACES}(\text{Ref}))$  as our coverage model;  $|\alpha(\text{TRACES}(\text{Ref}))| = 22,158$ . The choice of  $\ell$  here reflects the minimum number of transitions, in a  $n = 3$  system, required to reach all permissible concrete states from the initial state (III).

We provided simulation logs, that contain information to produce the set  $\text{TRACES}(\text{Ref})$ , to the monitor and collected coverage over our coverage model. The method for collecting coverage, similar to [4], involves counting the number



of times each coverage-point is observed. **Setup A** of Fig. 4 pictorially describes this experiment flow. After abstraction  $\alpha$  on the witness trace produced by the monitor, the process queries whether the abstracted trace exists in the coverage model. If it does, we mark that trace as covered. Since simulation logs for  $\text{TRACES}(Ref)$  are correct by construction, we would expect the monitor to have 100% coverage. This ensures that the witness traces produced by the monitor matches the abstract representatives of the inputs. However, we only had 35% coverage of traces of length 3. This means that the monitor produced vacuous pass results for the traces whose abstract representatives made up 65% of the coverage model. One of the main problems we have identified had to do with handling cacheline invalidation. Note that on an SMP environment, different systems may request ownership of a cacheline, but before a system acquires it, any outstanding shared-copy needs to be invalidated first. It turns out this feature accounts for more than 50% of the coverage model (recall that in the actual system, we have the  $|\Sigma_{\text{act}}| = 15$ , several of them relates to cache-invalidation, thus explaining the large impact on the coverage model).

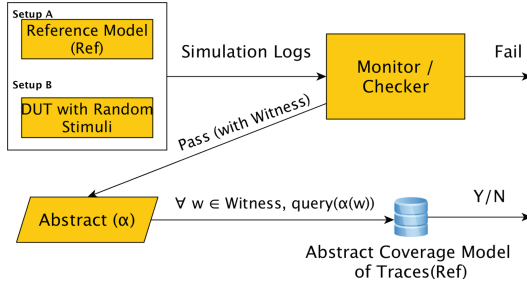
More importantly this experimental setup allows us to reason about the monitor’s trace-coverage on a *sequence* of operations on the cache. We were able to observe that the monitor had limited coverage on traces of length 2 or more that do not contain a particular subsequence  $cd$  where  $c, d \in \Sigma_{\text{act}}$  of type **Rd** (we obscure the specific operations  $c$  and  $d$  for IP non-disclosure), leading to the observation that the monitor was not handling such sequences correctly. It is promising that even with a small model we are already able to observe important holes in the monitor design. After fixing these problems, we were then confidently able to use the monitor on the test-bench with the IBM POWER<sup>TM</sup> Processor.

## 6.5 Experiments on a Large-Scale Verification Environment

In these experiments, we leverage a framework similar to the one described by Ludden et al. [18]. In other words, pseudorandom tests are generated [19], feeding the *DUT*. The *DUT* comprises of several cores, a multi-level cache hierarchy, and a data-interconnect. The communication traffic on the data-interconnect between different cores is snooped by the monitor, to perform certain checks.

We simulate this *DUT* model on 500 different random test-cases. This produces simulation logs which correspond to traces of arbitrary length on each cacheline (that is, each test-case may generate traces that exercise different cachelines). That is,  $|\text{TRACES}(DUT)| = 24,120$  each with arbitrary length. Using our framework (see **Setup B** of Fig. 4) we provided these simulation logs to the monitor and collected coverage. The coverage model  $\alpha(\text{TRACES}(Ref))$  in this experimental setup is obtained by accumulating the coverage models of *Ref* with  $n = 3$ , over lengths  $\ell = 1, 2, 3, 4$ . The resultant coverage model contains 278,715 coverage-points. We chose a shorter length for easier setup and diagnosis of the coverage information. Note that, the length of  $\ell$  can be configured as long as the resulting coverage model contains coverage-points that conform to the desired granularity of coverage analysis and tractability of model generation (the theoretical coverage model size grows exponentially with  $\ell$ ). The size of

the coverage model, determined by the choice of  $n$  and  $\ell$ , does not affect *DUT* simulation time since the analysis is post-simulation. Since the coverage tracking infrastructure is optimized to handle large coverage models, the performance overhead is minimal [18].



**Fig. 4.** Flow-diagram describing our coverage framework. **Setup A** is used for experiments in Sect. 6.4. **Setup B** is used for Sect. 6.5. If the monitor passes, then the procedure abstracts the *witness* traces and queries the pre-populated database for existence.

Given that the sample size of test-cases is only 500 (and  $|\text{TRACES}(DUT)| \ll 278,715$ ), we expect that the monitor’s coverage would be low compared to the size of the coverage model. Analyzing the monitor’s reports, we see that the monitor covered 437 unique entries of the coverage model. These entries map to 20,424 traces (85% of  $|\text{TRACES}(DUT)|$ ). That is, these 20,424 traces can be uniquely represented by 437 abstract traces. This represents a reduction of 46x w.r.t. a naive coverage model comprising of unabstracted traces. How about coverage-holes (the other 15%)? These can be broken down into two categories. The proposed abstraction ( $\alpha$ ) does not shorten the traces of the *DUT* to the length supported by the coverage model (accounting for 62%, or 2270, of them). In other words, the length of the abstracted trace remains greater than 4 after applying  $\alpha$ . To avoid these types of false coverage-holes one can either use a coverage model with a bigger length or use advanced rewriting on the abstracted trace (see Sect. 5-Corollary 1). Finally, the remaining 38% are due to *incorrect* test stimuli (that do not adhere to the specification and considered incorrect).

Our framework enables us to also make qualitative observations about the verification stimuli. For example, none of test-cases in our sample produce a trace containing an abstract state **SSS** nor **TSS**. This means that, there’s no single moment where all communicating nodes share a cacheline simultaneously. The distribution of the frequencies, that each coverage-point is hit, indicates that the test cases were heavily biased towards certain operational sequences on the cache (sample mean,  $\mu \approx 47$  and standard deviation,  $\sigma \approx 727$ ). In addition, we observe that in only about 4% of the traces, the *DUT* transitions from a single reader/writer epoch to a multiple reader epoch (and vice versa). Certainly, these under-exercised scenarios can lead to a design flaw. This is valuable feedback that

verification engineers can use to recalibrate the verification environment. Note that these types of operational sequences and their effect on the cache state are encoded by the coverage model in a way that is independent of the size of the *DUT*, leading to their reusability in the verification of larger *DUT*s.

## 7 Conclusion

We presented an abstraction approach as a novel framework for generating trace coverage models for symmetric systems. This methodology was validated using different sets of experiments while leveraging a large-scale verification environment. It demonstrated the method’s effectiveness even without utilizing tailored rewriting rules. We expect that future efforts will require this flexibility, and is a topic our next investigation.

The reasoning presented in this paper can be extended to other domains. In particular, some communication protocols with finite communicating windows may be very amenable to these techniques. Our formalization has a particular view of composed-systems that are abstractable, but this was motivated primarily to keep the presentation simple and concise. The abstraction involves “folding” that maintains information about one-or-more agent being in a particular state. Other abstractions may be defined that follow this coverage-model framework but require their own reasoning to establish Theorem 1. As an example, the state-abstraction approach that maintains a fixed number of agents [11] would lead to a trivial proof. We look forward to generalizing the approach to other such abstractions and variants of symmetric composed-systems.

**Acknowledgement.** The authors thank Viresh Paruthi and Jesse Bingham for valuable suggestions that helped with clarity of this paper.

## References

1. Miller, J.C., Maloney, C.J.: Systematic mistake analysis of digital computer programs. *Commun. ACM* **6**(2), 58–63 (1963)
2. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: *ICSE*, pp. 411–420. IEEE (1999)
3. Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: A temporal logic based theory of test coverage and generation. In: Katoen, J.-P., Stevens, P. (eds.) *TACAS 2002*. LNCS, vol. 2280, pp. 327–341. Springer, Heidelberg (2002). [https://doi.org/10.1007/3-540-46002-0\\_23](https://doi.org/10.1007/3-540-46002-0_23)
4. Ziv, A.: Cross-product functional coverage measurement with temporal properties-based assertions. In: *DATE*, p. 10834. IEEE (2003)
5. Chockler, H., Kupferman, O., Vardi, M.: Coverage metrics for formal verification. *STTT* **8**(4–5), 373–386 (2006)
6. Piziali, A.: *Functional Verification Coverage Measurement and Analysis*, 1st edn. Springer Publishing Company Incorporated, New York (2004). <https://doi.org/10.1007/b117979>
7. Czemerinski, H., Braberman, V., Uchitel, S.: Behaviour abstraction coverage as black-box adequacy criteria. In: *ICST*, pp. 222–231. IEEE (2013)

8. Castillos, K.C., Dadeau, F., Julliand, J.: Coverage criteria for model-based testing using property patterns. In: Proceedings of 9th MBT Workshop, pp. 29–43 (2014)
9. Papamarcos, M.S., Patel, J.H.: A low-overhead coherence solution for multiprocessors with private cache memories. In: Proceedings of 11th Annual International Symposium on Computer Architecture, pp. 348–354. ACM, New York (1984)
10. Shen, J., Abraham, J.A.: An RTL abstraction technique for processor microarchitecture validation and test generation. *J. Electron. Test.* **16**, 67–81 (2000)
11. Chou, Ching-Tsun, Mannava, Phanindra K., Park, Seungjoon: A simple method for parameterized verification of cache coherence protocols. In: Hu, Alan J., Martin, Andrew K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 382–398. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-30494-4\\_27](https://doi.org/10.1007/978-3-540-30494-4_27)
12. Book, R.V., Otto, F.: String-Rewriting Systems. Springer, New York (1993). <https://doi.org/10.1007/978-1-4613-9771-7>
13. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
14. Sinharoy, B., et al.: IBM POWER7 multicore server processor. *IBM J. Res. Dev.* **55**(3), 191–219 (2011)
15. Starke, W.J., et al.: The cache and memory subsystems of the IBM POWER8 processor. *IBM J. Res. Dev.* **59**(1), 3:1–3:13 (2015)
16. Cragon, H.G.: Memory Systems and Pipelined Processors. Jones and Bartlett Publishers, Burlington (1996)
17. Shimizu, K., et al.: Verification of the cell broadband engine; processor. In: Proceedings of 43rd Annual DAC, pp. 338–343. ACM (2006)
18. Ludden, J.M., et al.: Functional verification of the POWER4 microprocessor and POWER4 multiprocessor system. *IBM J. Res. Dev.* **46**(1), 53–76 (2002)
19. Adir, A., et al.: Genesys-pro: innovations in test program generation for functional processor verification. *IEEE Des. Test Comput.* **21**(2), 84–93 (2004)



# Verification of Fault-Tolerant Protocols with Sally

Bruno Dutertre<sup>(✉)</sup>, Dejan Jovanović<sup>(✉)</sup>, and Jorge A. Navas<sup>(✉)</sup>

Computer Science Laboratory, SRI International, Menlo Park, USA  
{bruno.dutertre,dejan.jovanovic,jorge.navas}@sri.com

**Abstract.** Sally is a model checker for infinite-state systems that implements several verification algorithms, including a variant of IC3/PDR called Property-Directed K-induction. We present an application of Sally to automated verification of fault-tolerant distributed algorithms.

## 1 Introduction

Sally is a new model checker for infinite-state systems developed by SRI International. It is a successor of the Symbolic Analysis Laboratory (SAL) [6]. Sally supports bounded model checking and proof by  $k$ -induction, and it implements a novel model-checking algorithm based on IC3/PDR that can automatically discover  $k$ -inductive strengthening of a property of interest. Details of this *Property-Directed K-induction* (PD-KIND) algorithm are presented in [12].

We present an application of Sally to fault-tolerant distributed algorithms. We focus on a class of synchronous algorithms that consist of one or more rounds of communication between  $N$  processes—some of which may be faulty—followed by some form of averaging or voting to achieve agreement among processes. This type of algorithm is at the core of many fault-tolerant systems used in avionics or other control systems, including protocols for fault-tolerant sampling of sensor data and clock-synchronization protocols. Until the advent of PDR and relatives, such protocols could not be verified automatically by model checkers. The best technique available was  $k$ -induction, which is typically not fully automatic and requires expertise to discover auxiliary inductive invariants. We show that PD-KIND can automatically verify complex fault-tolerant algorithms, under a variety of fault assumptions.

## 2 Sally

Sally is a modular and extensible framework for prototyping and development of model-checking algorithms. Currently, Sally implements several algorithms

---

This work was supported in part by NASA Cooperative Agreement NNX14AI05A and by NSF grant 1528153. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

based on satisfiability modulo theories (SMT), including bounded model checking and  $k$ -induction, and the novel PD-KIND algorithm [12]. PD-KIND generalizes IC3/PDR [4, 10] by relying on  $k$ -induction and  $k$ -step reachability as subprocedures, rather than ordinary one-step induction and reachability. The PD-KIND procedure relies on backend SMT solvers to provide features such as model-based generalization [13] and interpolants. The current implementation combines Yices 2 [8] and MathSAT 5 [5]. Other backend solvers can also be used for  $k$ -induction and bounded model checking. Sally is open source software available at <https://github.com/SRI-CSL/sally>.

The primary input language of Sally is called MCMT (for Model Checking Modulo Theories). This language extends the SMT-LIB 2 standard [1] with commands for defining transition systems. SMT-LIB 2 is used to represent terms and formulas. Transition systems are defined by specifying a state space, a set of initial states, and a transition relation. MCMT also allows one to specify invariant properties. Sally can parse other input languages than MCMT and internally convert them to MCMT. All our examples are written in a subset of the SAL language [7], converted to MCMT, then analyzed using Sally.

### 3 Modeling Fault-Tolerant Protocols

We use a simple modeling approach that is generally applicable to synchronous algorithms. The system state is a finite set of arrays indexed by process identities. Communication channels are also modeled using arrays (e.g., a channel from process  $i$  to process  $j$  is represented as an array element  $c[i][j]$ ). Each transition of the system corresponds to one round of the algorithm: a process  $i$  updates its local variables then send data on one or more communication channel.

To model faults, we assign a status to each process and we specify faulty behavior as assumptions on the data transmitted by processes. A faulty process is then assumed to execute the algorithm correctly, except when it sends data. This approach simplifies process specifications and is sufficient for all types of process faults [20].

#### 3.1 Approximate Agreement

We illustrate our approach using a protocol based on the unified fault-tolerant protocol of Miner et al. [16]. The protocol ensures approximate agreement. It assumes inexact communication, which models errors in sensor sampling or clock drifts. The fault model distinguishes between omissive and transmissive faults, and between symmetric and asymmetric behavior. A symmetric omissive process either fails to send data (on all its channels) or sends correct data. An asymmetric omissive process may send nothing to some and correct data to other processes. A symmetric transmissive process sends possibly incorrect data, but it sends the same data on all its channels. An asymmetric transmissive process behaves in an arbitrary way.

The protocol involves  $N$  processes. Process  $i$  holds a real value  $v[i]$ . In each round, this process broadcasts its value to the all processes,<sup>1</sup> computes a fault-tolerant average of the values it receives, and updates  $v[i]$  using this average. The protocol is intended to ensure *convergence*: the absolute difference between  $v[i]$  and  $v[j]$  is approximately reduced by half with each protocol round.

We model this protocol as a single state-transition system that operates on arrays. The main state variables include an array  $v$  that stores process values, and arrays  $m$  and  $c$  that model communication channels:

```
v: ARRAY PID OF DATA,
m: ARRAY PID OF ARRAY PID OF BOOLEAN,
c: ARRAY PID OF ARRAY PID OF DATA,
```

Variable  $m[j][i]$  indicates that the message from  $i$  to  $j$  is missing. If  $m[j][i]$  is true, then  $c[j][i]$  is ignored, otherwise  $c[j][i]$  is the value that  $j$  receives from  $i$ . We formalize the fault model as constraints on  $m'[j][i]$  and  $c'[j][i]$  based on the status of process  $i$ . These constraints are as follows:<sup>2</sup>

```
(FORALL (i: PID): status[i] = Good =>
  (FORALL (j: PID): NOT m'[j][i] AND received(v[i], c'[j][i])))

(FORALL (i: PID): status[i] = SymmetricOmissive =>
  (FORALL (j: PID): m'[j][i] OR (FORALL (j: PID): received(v[i], c'[j][i])))

(FORALL (i: PID): status[i] = AsymmetricOmissive =>
  (FORALL (j: PID): m'[j][i] OR received(v[i], c'[j][i])))

(FORALL (i: PID): status[i] = SymmetricTransmissive =>
  (FORALL (j: PID): m'[j][i]
    OR (FORALL (j, k: PID): c'[j][i] - c'[k][i] <= 2 * epsilon)))
```

The parameter  $\epsilon$  is a bound on communication error; if a process sends a value  $x$  then the recipient reads a value in the interval  $[x - \epsilon, x + \epsilon]$ . In the above rules, this communication error is specified by predicate *received*:

```
received(x: DATA, y: DATA): BOOLEAN = x - epsilon <= y AND y <= x + epsilon;
```

A non-trivial part of the model is the definition of the fault-tolerant average. We use a form of mid-value select, parameterized by an constant  $\tau$ : when a process  $i$  receives  $n \leq N$  values in round  $k$ , it sorts these values in increasing order to form a sequence of reals  $x_1, \dots, x_n$ . The mid-value select is the average of  $x_{\tau+1}$  and  $x_{n-\tau}$ . (If  $n < \tau$ , a default value is chosen.) In practice, the parameter  $\tau$  is equal to the number of asymmetric faults to tolerate, and must be chosen so that  $n > 2\tau$ .

We do not want to write a sorting algorithm in SAL, as translation to MCMT requires all functions applications to be inlined. For any sorting algorithm, this unrolling inevitably would cause an exponential blowup. Instead, we use a specification trick. We introduce two auxiliary state variables  $p$  and  $n$ :

```
p: ARRAY PID OF ARRAY PID OF PID,
n: ARRAY PID OF [0 .. N],
```

<sup>1</sup> To avoid special cases, we assume that  $i$  is included in the set of recipients.

<sup>2</sup> The actual SAL syntax is less readable but equivalent.

For a process  $i$ ,  $n[i]$  denotes the number of messages received by  $i$ , and  $p[i]$  is a permutation of the indices in  $\{1, \dots, N\}$  that enumerates the  $n$  received values in increasing order. We specify these relations as shown in Fig. 1. This essentially states the post-condition of the sorting algorithm we need. The input is an array  $v$  of  $N$  values and an array  $m$  of Boolean flags; where  $m[i]$  true means that  $v[i]$  is missing. The output includes a variable  $n$  that counts the number of non-missing elements, and a permutation  $p$  that sorts the non-missing elements in increasing order. From  $p$ ,  $n$ , and  $v$ , we can easily define the mid-value select.

```

sort_and_filter(v: ARRAY PID OF DATA,
               m: ARRAY PID OF BOOLEAN,
               n: [0 .. N],
               p: ARRAY PID OF PID): BOOLEAN =
  (FORALL (i: PID): (i < n => m[p[i]]))
AND (FORALL (i: PID): i < n => v[p[i]] <= v[p[i+1]])
AND (FORALL (i, j: PID): p[i] = p[j] => i = j);

```

Fig. 1. Sorting and filter predicate

The final step is to specify the convergence property. The values  $v[i]$ s are initially within some distance  $\Delta$  of each other and get closer and closer with each protocol round. Because of the communication error, the best bound one can achieve is  $2\epsilon$ . The protocol converges towards this bound at an exponential rate. A more precise specification is shown in Fig. 2. We add a state variable `delta` to our state-transition system to store the bound. The variable is initialized to an arbitrary bound larger than  $2\epsilon$ , then it is updated with every protocol round as shown in the figure. Our goal is to show that the `convergence` property is invariant: the difference between  $v[i]$  and  $v[j]$  is bounded by `delta`.

```

% Initial precision: maximum difference between the initial values
initial_delta: { x: REAL | x > 2 * epsilon };

% Convergence function: if all values are within some delta
% at round k then they are within next(delta) at round k+1.
next(x: REAL): REAL = x/2 + 2 * epsilon;

% Precision improvement with each round
delta' = next(delta);

% Convergence property for the approx system
convergence: LEMMA approx |- G(FORALL (i, j: PID): v[i] - v[j] <= delta);

```

Fig. 2. Convergence and approximate agreement property

### 3.2 Verification Results

We have analyzed the approximate agreement protocol under four scenarios, and for different values of  $N$ . Each scenario makes different fault assumptions:



no faults (Scenario 0); one symmetric transmissive and one asymmetric omissive faults (Scenario 1); one asymmetric transmissive and one asymmetric omissive faults (Scenario 2); and one asymmetric transmissive, one asymmetric omissive, and one symmetric omissive faults (Scenario 3). The results are summarized in Table 1. The left part shows the results and runtime of Sally’s  $k$ -induction engine. The right-hand side shows results and runtimes of Sally’s PD-KIND. The  $k$ -induction engine iteratively tries  $k$ -induction for  $k$  from 1 to 10.

**Table 1.** Analysis results. Each entry reports the result and runtime of an experiment. v means *valid* (the property was proved), i means *invalid* (a counterexample was produced), u means *unknown* ( $k$ -induction was inconclusive), t means timeout. Runtimes are CPU time in seconds. The timeout is 5000 s.

N	K-induction				PD-Kind			
	Scen. 0	Scen. 1	Scen. 2	Scen. 3	Scen. 0	Scen. 1	Scen. 2	Scen. 3
4	u 88	i 0	i 0	i 0	v 3	i 0	i 0	i 0
5	t -	t -	t -	i 0	v 57	v 84	v 47	i 1
6	t -	t -	t -	t -	v 397	v 1742	v 1771	v 461
7	t -	t -	t -	t -	v 3581	v 3935	v 4838	v 3124

The convergence property is not  $k$ -inductive, so  $k$ -induction cannot prove it. On the other hand, for instances where the property does not hold, then the  $K$ -induction engine finds a counterexample very quickly. PD-KIND works much better. On all instances where the property holds, it can automatically prove it. On all instances where the property is false, it can find a counterexample. The verification cost tends to be higher with more complex fault models. Because the algorithm is quite complex, scalability is an issue. The runtime grows very quickly as  $N$  increases, both for the PD-KIND and  $k$ -induction engines. We believe the complexity of the averaging function is the main bottleneck for this example.

We have verified simpler fault-tolerant algorithm such as OM1, which uses majority voting. Our formalization is based on the Boyer-Moore algorithm [3]. Table 2 shows runtimes for a variant of OM1 that uses  $N$  processes and  $M$  relays. We used Sally to prove the two classic properties of OM1—agreement (**P1**) and validity (**P2**)—in a scenario with one Byzantine-faulty relay. It turns out that both properties are  $k$ -inductive. As shown in the table, the  $k$ -induction engine proves the properties in a few seconds at most. PD-KIND also works for these examples, but it is much slower. For example, proving agreement for  $N = 5$  and  $M = 20$  takes about 4 sec for  $k$ -induction and more than 20 min for PD-KIND.

## 4 Related Work

Developing correct distributed algorithms is notoriously hard; making sure that these algorithms tolerate failures is even harder. Since the 1980s, formal methods

**Table 2.** Runtimes on variants of the OM1 protocol (seconds of CPU time).

		<i>M</i>																
		<i>N</i>	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
<b>P1</b>	3	0.04	0.05	0.06	0.08	0.09	0.11	0.21	0.18	0.24	0.32	0.46	0.43	0.53	0.85	0.85	0.91	
	4	0.06	0.08	0.13	0.16	0.21	0.22	0.30	0.41	0.56	0.61	0.74	0.67	1.04	0.94	1.47	1.66	
	5	0.09	0.15	0.21	0.27	0.37	0.54	0.79	0.77	0.86	1.11	1.08	1.31	1.87	2.55	3.11	3.36	
<b>P2</b>	3	0.03	0.04	0.06	0.08	0.11	0.13	0.14	0.18	0.30	0.36	0.44	0.48	0.59	0.85	0.94	1.04	
	4	0.06	0.10	0.13	0.16	0.30	0.28	0.40	0.35	0.47	0.78	1.02	0.83	1.02	1.17	1.77	1.61	
	5	0.09	0.13	0.20	0.38	0.37	0.47	0.49	0.70	0.88	1.20	1.14	1.65	2.02	2.28	2.73	3.38	

have been used to precisely model and mathematically prove the correctness of such fault-tolerant algorithms. Most of this work use interactive theorem provers (e.g., [15, 17, 18, 21]). More recently, Padon et al. use a semi-automated proof method based encoding protocol rules into a decidable logic [19].

Model checking using abstraction technique has also been applied to this domain. Konnov and his colleagues show how a threshold-based algorithms can be modeled using counter systems, and develop verification algorithms for these systems [11, 14]. An earlier example by Fisman et al. [9] uses another abstraction technique and applies regular model checking [2]. These abstraction methods are limited to special classes of protocols. The type of algorithms that we have presented manipulate numerical data and rely on non-trivial computation, and do not belong to these classes. When abstractions are not applicable, proofs using  $k$ -induction are possible but such proofs can be difficult, and require expertise to identify key auxiliary invariants.

## 5 Conclusion

Until recently, automated verification of complex fault-tolerant algorithms was impossible. One either had to resort to interactive theorem proving—which is slow and requires expertise—or rely on semi-automated method such as  $k$ -induction. New model-checking algorithms based on IC3/PDR have changed the picture; it is now feasible to verify a rich class of fault-tolerant protocols in a fully automated manner. Remaining challenges include improving scalability of these methods and extending them to richer logical theories.

## References

1. Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-LIB standard: Version 2.0
2. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000). [https://doi.org/10.1007/10722167\\_31](https://doi.org/10.1007/10722167_31)
3. Boyer, R.S., Moore, J.S.: MJRTY—a fast majority vote algorithm. In: Boyer, R.S. (ed.) Automated Reasoning: Essays in Honor of Woody Blesdole, vol. 1, pp. 105–117. Springer, Dordrecht (1991). [https://doi.org/10.1007/978-94-011-3488-0\\_5](https://doi.org/10.1007/978-94-011-3488-0_5)

4. 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](https://doi.org/10.1007/978-3-642-18275-4_7)
5. 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](https://doi.org/10.1007/978-3-642-36742-7_7)
6. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 496–500. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-27813-9\\_45](https://doi.org/10.1007/978-3-540-27813-9_45)
7. de Moura, L., Owre, S., Shankar, N.: The SAL language manual. Technical Report SRI-CSL-01-02, Computer Science Laboratory, SRI International (2003)
8. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-08867-9\\_49](https://doi.org/10.1007/978-3-319-08867-9_49)
9. Fisman, D., Kupferman, O., Lustig, Y.: On verifying fault tolerance of distributed protocols. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 315–331. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-78800-3\\_22](https://doi.org/10.1007/978-3-540-78800-3_22)
10. 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](https://doi.org/10.1007/978-3-642-31612-8_13)
11. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In: FMCAD, pp. 201–209 (2013)
12. Jovanović, D., Dutertre, B.: Property-directed k-induction. In: FMCAD, pp. 85–92 (2016)
13. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 17–34. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-08867-9\\_2](https://doi.org/10.1007/978-3-319-08867-9_2)
14. Konnov, I., Veith, H., Widder, J.: SMT and POR beat counter abstraction: parameterized model checking of threshold-based distributed algorithms. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 85–102. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-21690-4\\_6](https://doi.org/10.1007/978-3-319-21690-4_6)
15. Lincoln, P., Rushby, J.: Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In: COMPASS, pp. 107–120 (1994)
16. Miner, P., Geser, A., Pike, L., Maddalon, J.: A unified fault-tolerance protocol. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT -2004. LNCS, vol. 3253, pp. 167–182. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-30206-3\\_13](https://doi.org/10.1007/978-3-540-30206-3_13)
17. Miner, P.S.: Verification of fault-tolerant clock synchronization systems. NASA Technical Paper 3349 (1993)
18. Owre, S., Rushby, J., Shankar, N., von Henke, F.: Formal verification for fault-tolerant architectures: prolegomena to the design of PVS. IEEE Trans. Softw. Eng. **21**(2), 107–125 (1995)
19. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made EPR: decidable reasoning about distributed protocols. In: OOPSLA, vol. 1, pp. 108:1–108:31 (2017)

20. Pike, L., Maddalon, J., Miner, P., Geser, A.: Abstractions for fault-tolerant distributed system verification. In: Slind, K., Bunker, A., Gopalakrishnan, G. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 257–270. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-30142-4\\_19](https://doi.org/10.1007/978-3-540-30142-4_19)
21. Wilcox, J.R., Woos, D., Pancheckha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: a framework for implementing and formally verifying distributed systems. In: PLDI, pp. 357–368 (2015)



# Output Range Analysis for Deep Feedforward Neural Networks

Souradeep Dutta<sup>1</sup> , Susmit Jha<sup>2</sup> , Sriram Sankaranarayanan<sup>1</sup>  ,  
and Ashish Tiwari<sup>2</sup> 

<sup>1</sup> University of Colorado, Boulder, USA

{souradeep.dutta,sriram.sankaranarayanan}@colorado.edu

<sup>2</sup> SRI International, Menlo Park, USA

{susmit.jha,tiwari}@cs1.sri.com

**Abstract.** Given a neural network (NN) and a set of possible inputs to the network described by polyhedral constraints, we aim to compute a safe over-approximation of the set of possible output values. This operation is a fundamental primitive enabling the formal analysis of neural networks that are extensively used in a variety of machine learning tasks such as perception and control of autonomous systems. Increasingly, they are deployed in high-assurance applications, leading to a compelling use case for formal verification approaches. In this paper, we present an efficient range estimation algorithm that iterates between an expensive global combinatorial search using mixed-integer linear programming problems, and a relatively inexpensive local optimization that repeatedly seeks a local optimum of the function represented by the NN. We implement our approach and compare it with Reluplex, a recently proposed solver for deep neural networks. We demonstrate applications of our approach to computing flowpipes for neural network-based feedback controllers. We show that the use of local search in conjunction with mixed-integer linear programming solvers effectively reduces the combinatorial search over possible combinations of active neurons in the network by pruning away suboptimal nodes.

## 1 Introduction

Deep neural networks have emerged as a versatile and popular representation for machine learning models. This is due to their ability to approximate complex functions, as well as the availability of efficient methods for learning these from large data sets. The black box nature of NN models and the absence of effective methods for their analysis has confined their use in systems with low integrity requirements. However, more recently, deep NNs are also being adopted in high-assurance systems, such as automated control and perception pipeline of autonomous vehicles [13] or aircraft collision avoidance [12]. While traditional system design approaches include rigorous system verification and analysis techniques to ensure the correctness of systems deployed in safety-critical applications [1], the inclusion of complex machine learning models in the form of deep

NNs has created a new challenge to verify these models. In this paper, we focus on the *range estimation problem*, wherein, given a neural network  $N$  and a polyhedron  $\phi(\mathbf{x})$  representing a set of inputs to the network, we wish to estimate a range, denoted as  $\mathbf{range}(l_i, \phi)$ , for each of the network’s output  $l_i$  that subsumes all possible outputs and is tight within a given tolerance  $\delta$ . We restrict our attention to feedforward deep NNs. While we focus on NNs that use rectified linear units (ReLUs) [17] as activation functions, we also discuss extensions to other activation functions through piecewise linear approximations.

Our approach is based on augmenting a mixed-integer linear programming (MILP) solver. First of all, we use a sound piecewise linearization of the nonlinear activation function to define an encoding of the neural network semantics into mixed-integer constraints involving real-valued variables and binary variables that arise from the (piecewise) linearized activation functions. The encoding into MILP is a standard approach to handling piecewise linear functions [28]. As such, the input constraints  $\phi(\mathbf{x})$  are added to the MILP and next, the output variable is separately maximized and minimized to infer a range. Our approach combines the MILP solver with a local search that exploits the local continuity and differentiability properties of the function represented by the network. These properties are not implicit in the MILP encoding that typically relies on a branch-and-cut approach to solve the problem at hand. On the other hand, local search alone may get “stuck” in local minima. Our approach handles local minima by using the MILP solver to search for a solution that is “better” than the current local minimum or conclude that no such solution exists. Thus, by alternating between inexpensive local search iterations and relatively expensive MILP solver calls, we seek an approach that can exploit local properties of the neural network function but at the same time avoid the problem of local minima.

The range estimation problem has several applications. For instance, a safety focused application of the range estimation problem arises when we have deep neural networks implementing a controller. In this case, the range estimation problem enables us to prove bounds on the output of the NN controller. This is important because out-of-bounds outputs can drive the physical system into undesirable configurations, such as the locking of robotic arm, or command a car’s throttle beyond its rated limits. Finding these errors through verification will enable design-time detection of potential failures instead of relying on run-time monitoring which can have significant overhead and also may not allow graceful recovery. Additionally, range analysis can be useful in proving the safety of a closed loop system by integrating the action of a neural network controller with that of a plant model. In this paper, we focus on the application of range estimation problem to proving safety of several neural network plant models along with neural network feedback controllers. Other applications include proving the robustness of classifiers by showing that all possible input perturbations within some range do not change the output classification of the network.

**Related Work.** The importance of analytical certification methods for neural networks has been well-recognized in literature. Neural networks have been

observed to be very sensitive to slight perturbations in their inputs producing incorrect outputs [21, 26]. This creates a pressing need for techniques to provide formal guarantees on the neural networks. The verification of neural networks is a hard problem, and even proving simple properties about them is known to be NP-complete [14]. The complexity of verifying neural networks arises primarily from two sources: the nonlinear activation functions used in the network as elementary neural units and the structural complexity that can be measured using depth and size of the network. Kurd [16] presented one of the first categorization of verification goals for NNs used in safety-critical applications. The proposed approach here targets a subset of these goals, G4 and G5, which aim at ensuring robustness of NNs to disturbances in inputs, and ensuring the output of NNs are not hazardous.

Recently, there has been a surge of interest in formal verification tools for neural networks [8, 10, 14, 18, 22, 23, 25, 30, 31]. A detailed discussion of these approaches to neural networks with piecewise linear activation functions, and empirical evaluations over benchmark networks has been carried out by Bunel et al. [5]. Our approach relies on a piecewise linearization of the nonlinear activation function. This idea has been studied in the past, notably by Pulina et al. [22, 23]. The key differences include: (a) our approach do not perform a refinement operation. As such, no refinement is needed for networks with piecewise linear activation functions, since the activation functions are encoded precisely. For other kinds of functions such as sigmoid or tanh, a refinement may be needed to improve the inferred ranges, but is not considered in our work. (b) We do not rely on existing Satisfiability-Modulo Theory (SMT) solvers [2]. Instead, our approach uses a mixed-integer linear programming (MILP) solver in combination with a local search. Recently, Lomuscio and Maganti present an approach that encodes neural networks into MILP constraints [18]. A similar encoding is also presented by Tjeng and Tedrake [27] for verifying robustness of neural network classifiers under a class of perturbations. These encodings are similar to ours. The optimization problems are solved directly using an off-the-shelf MILP solver [4, 28]. Additionally, our approach augments the MILP solver with a local search scheme. We note that the use of local search can potentially speed up our approach, since neural networks represent continuous, piecewise-differentiable functions. On the flip side, these functions may have a large number of local minima/maxima. Nevertheless, depending on the network, the function it approximates and the input range, the local search used in conjunction with a MILP solver can yield rapid improvements to the objective function.

Augmenting existing LP solvers has been at the center of two recent approaches to the problem. The Reluplex approach by Katz et al. focuses on ReLU feed-forward networks [14]. Their work augments the Simplex algorithm with special functions and rules that handle the constraints involving ReLU activation functions. The linear programming used for comparison in Reluplex performs significantly less efficiently according to the experiments reported in this paper [14]. Note, however, that the scenarios used by Katz et al. are different from those studied here, and were not publicly available for comparison at the

time of writing. Ehlers augments a LP solver with a SAT solver that maintains partial assignments to decide the linear region for each individual neuron. The solver is instantiated using facts inferred from a convexification of the activation function [8], much in the style of conflict clauses and lemmas used by SAT solvers. In fact, many ideas used by Ehlers can be potentially used to complement our approach in the form of cuts that are specific to neural networks. Such specialized cuts are very commonly used in MILP solvers.

A related goal of finding adversarial inputs for deep NNs has received a lot of attention, and can be viewed as a testing approach to NNs instead of verification method discussed in this paper. A linear programming based approach for finding adversarial inputs is presented in [3]. A related approach for finding adversarial inputs using SMT solvers that relies on a layer-by-layer analysis is presented in [10]. Simulation-based approaches [30] for neural network verification have also been proposed in literature. This relies on turning the reachable set estimation problem into a neural network maximal sensitivity computation, and solving it using a sequence of convex optimization problems. In contrast, our proposed approach combines numerical gradient-based optimization with mixed-integer linear programming for more efficient verification.

**Contributions.** We present a novel algorithm for propagating convex polyhedral inputs through a feedforward deep neural network with ReLU activation units to establish ranges for the outputs of the network. We have implemented our approach in a tool called SHERLOCK [6]. We compare SHERLOCK with a recently proposed deep NN verification engine - Reluplex [14]. We demonstrate the application of SHERLOCK to establish output range of deep NN controllers. Our approach seems to scale *consistently* to neural networks having 100 neurons to as many as over 6000 neurons.

## 2 Preliminaries

We present the preliminary notions including deep neural networks, polyhedra, and mixed integer linear programs.

We will study feed forward neural networks (NN) throughout this paper with  $n > 0$  inputs and  $m > 0$  outputs. For simplicity, we will present our techniques primarily for the single output case ( $m = 1$ ), explaining how they can be extended to networks with multiple outputs.

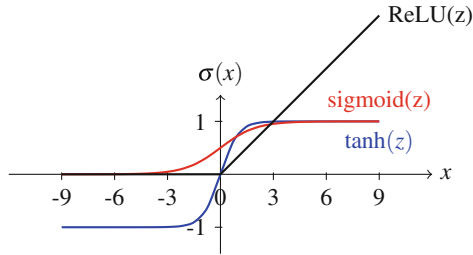
Let  $\mathbf{x} \in \mathbb{R}^n$  denote the inputs and  $y \in \mathbb{R}$  be the output of the network. Structurally, a NN  $\mathcal{N}$  consists of  $k > 0$  hidden layers, wherein we assume that each layer has the same number of neurons  $N > 0$ . We use  $N_{ij}$  to denote the  $j^{\text{th}}$  neuron of the  $i^{\text{th}}$  layer for  $j \in \{1, \dots, N\}$  and  $i \in \{1, \dots, k\}$ .

**Definition 1 (Neural Network).** A  $k$  layer neural network with  $N$  neurons per hidden layer is described by matrices:  $(W_0, \mathbf{b}_0), \dots, (W_{k-1}, \mathbf{b}_{k-1}), (W_k, \mathbf{b}_k)$ , wherein (a)  $W_0, \mathbf{b}_0$  are  $N \times n$  and  $N \times 1$  matrices denoting the weights connecting the inputs to the first hidden layer, (b)  $W_i, \mathbf{b}_i$  for  $i \in [1, k-1]$  connect layer  $i$  to layer  $i+1$  and (c)  $W_k, \mathbf{b}_k$  connect the last layer  $k$  to the output.



Each neuron is defined using its *activation function*  $\sigma$  linking its input value to the output value. Although this can be any function, there are a few common activation functions:

1. **ReLU:** The ReLU unit is defined by the activation function  $\sigma(z) : \max(z, 0)$ .
2. **Sigmoid:** The sigmoid unit is defined by the activation function  $\sigma(z) : \frac{1}{1+e^{-z}}$ .
3. **Tanh:** The activation function for this unit is  $\sigma(z) : \tanh(z)$ .



**Fig. 1.** Activation functions commonly used in neural networks.

Figure 1 shows these functions graphically. We will assume that all the neurons of the network  $\mathcal{N}$  have the same activation function  $\sigma$ . Furthermore, we assume that  $\sigma$  is a continuous function and differentiable almost everywhere.

Given a neural network  $\mathcal{N}$  as described above, the function  $F : \mathbb{R}^n \rightarrow \mathbb{R}$  computed by the neural network is given by the composition  $F := F_k \circ \dots \circ F_0$  wherein  $F_i(\mathbf{z}) : \sigma(W_i \mathbf{z} + \mathbf{b}_i)$  is the function computed by the  $i^{\text{th}}$  hidden layer,  $F_0$  the function linking the inputs to the first layer, and  $F_k$  linking the last layer to the output.

For a fixed input  $\mathbf{x}$ , it is easily seen that the function  $F$  computed by a NN  $\mathcal{N}$  is continuous and nonlinear, due to the activation function  $\sigma$ . For the case of neural networks with ReLU units, this function is piecewise affine, and differentiable almost everywhere in  $\mathbb{R}^n$ . For smooth activation functions such as tanh and sigmoid, the function is differentiable as well. If it exists, we denote the gradient of this function  $\nabla F : (\partial_{x_1} F, \dots, \partial_{x_n} F)$ . Computing the gradient can be performed efficiently (as described subsequently).

## 2.1 Mixed Integer Linear Programs

Throughout this paper, we will formulate linear optimization problems with integer variables. We briefly recall these optimization problems, their computational complexity and solution techniques used in practice.

**Definition 2 (Mixed Integer Program).** *A mixed integer linear program (MILP) involves a set of real-valued variables  $\mathbf{x}$  and integer valued variables  $\mathbf{w}$  of the following form:*

$$\begin{aligned} \max \quad & \mathbf{a}^T \mathbf{x} + \mathbf{b}^T \mathbf{w} \\ \text{s.t.} \quad & A\mathbf{x} + B\mathbf{w} \leq \mathbf{c} \\ & \mathbf{x} \in \mathbb{R}^n, \mathbf{w} \in \mathbb{Z}^m \end{aligned}$$

The problem is called a linear program (LP) if there are no integer variables  $\mathbf{w}$ . The special case wherein  $\mathbf{w} \in \{0, 1\}^m$  is called a binary MILP. Finally, the case without an explicit objective function is called an MILP feasibility problem.

It is well known that MILPs are NP-hard problems: the best known algorithms, thus far, have exponential time worst case complexity. We will later briefly review the popular branch-and-cut class of algorithms for solving MILPs at a high level. These algorithms along with the associated heuristics underlie highly successful, commercial MILP solvers such as Gurobi [9] and CPLEX [11].

### 3 Problem Definition and MILP Encoding

Let  $\mathcal{N}$  be a neural network with inputs  $\mathbf{x} \in \mathbb{R}^n$ , output  $y \in \mathbb{R}$  and weights  $(W_0, \mathbf{b}_0), \dots, (W_k, \mathbf{b}_k)$ , activation function  $\sigma$  for each neuron unit, defining the function  $F_N : \mathbb{R}^n \rightarrow \mathbb{R}$ .

**Definition 3 (Range Estimation Problem).** *The problem is defined as follows:*

- INPUTS: Neural network  $\mathcal{N}$ , and input constraints  $P : A\mathbf{x} \leq \mathbf{b}$  that is compact: i.e., closed and bounded in  $\mathbb{R}^n$ . A tolerance parameter is a real number  $\delta > 0$ .
- OUTPUT: An interval  $[\ell, u]$  such that  $(\forall \mathbf{x} \in P) F_N(\mathbf{x}) \in [\ell, u]$ . I.e.,  $[\ell, u]$  contains the range of  $F_N$  over inputs  $\mathbf{x} \in P$ . Furthermore, the interval is  $\delta$ -tight:

$$u - \delta \leq \max_{\mathbf{x} \in P} F_N(\mathbf{x}) \text{ and } \ell + \delta \geq \min_{\mathbf{x} \in P} F_N(\mathbf{x}).$$

Without loss of generality, we will focus on estimating the upper bound  $u$ . The case for the lower bound will be entirely analogous.

#### 3.1 MILP Encoding

We will first describe the MILP encoding when  $\sigma$  is defined by a ReLU unit. The treatment of more general activation functions will be described subsequently. The real-valued variables of the MILP are as follows:

1.  $\mathbf{x} \in \mathbb{R}^n$ : the inputs to the network with  $n$  variables.
2.  $\mathbf{z}_1, \dots, \mathbf{z}_{k-1}$ , the outputs of the hidden layer. Each  $\mathbf{z}_i \in \mathbb{R}^N$ .
3.  $y \in \mathbb{R}$ : the overall output of the network.

Additionally, we introduce binary (0/1) variables  $\mathbf{t}_1, \dots, \mathbf{t}_{k-1}$ , wherein each vector  $\mathbf{t}_i \in \mathbb{Z}^N$  (the same size as  $\mathbf{z}_i$ ). These variables will be used to model the piecewise behavior of the ReLU units.

Next, we encode the constraints. The first set of constraints ensure that  $\mathbf{x} \in P$ . Suppose  $P$  is defined as  $A\mathbf{x} \leq \mathbf{b}$  then we simply add the constraints  $C_0 : A\mathbf{x} \leq \mathbf{b}$ .

For each hidden layer  $i$ , we require that  $\mathbf{z}_{i+1} = \sigma(W_i\mathbf{z}_i + \mathbf{b}_i)$ . Since  $\sigma$  is not linear, we use the binary variables  $\mathbf{t}_{i+1}$  to encode the same behavior:

$$C_{i+1} : \begin{cases} \mathbf{z}_{i+1} \geq W_i\mathbf{z}_i + \mathbf{b}_i, \\ \mathbf{z}_{i+1} \leq W_i\mathbf{z}_i + \mathbf{b}_i + M\mathbf{t}_{i+1}, \\ \mathbf{z}_{i+1} \geq 0, \\ \mathbf{z}_{i+1} \leq M(\mathbf{1} - \mathbf{t}_{i+1}) \end{cases}$$

Note that for the first hidden layer, we simply substitute  $\mathbf{x}$  for  $\mathbf{z}_0$ . This trick of using binary variables to encode piecewise linear function is standard in optimization [28, Chap. 22.4] [29, Chap. 9]. Here  $M$  needs to be larger than the maximum possible output at any node. We can derive fast estimates for  $M$  through interval analysis by using the norms  $\|W_i\|_\infty$  and the bounding box of the input polyhedron.

The output  $y$  is constrained as:  $C_{k+1} : y = W_k\mathbf{z}_k + \mathbf{b}_k$ .

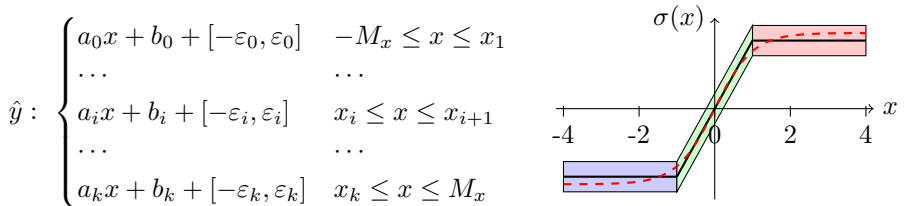
The MILP, obtained by combining these constraints, is of the form:

$$\begin{aligned} \max y \text{ s.t. constraints } C_0, \dots, C_{k+1} (\text{see above}) \\ \mathbf{x}, \mathbf{z}_1, \dots, \mathbf{z}_k, y \in \mathbb{R}^{kN+n+1} \\ \mathbf{t}_1, \dots, \mathbf{t}_{k-1} \in \mathbb{Z}^{(k-1)N} \end{aligned} \tag{3.1}$$

**Theorem 1.** *The MILP encoding in (3.1) is always feasible and bounded. Its optimal solution  $u^*$  corresponds to an input to the network  $\mathbf{x}^* \in P$  such that  $y = F_N(\mathbf{x}) = u^*$ . Furthermore, for all  $\mathbf{x} \in P$ ,  $F_N(\mathbf{x}) \leq u^*$ .*

*Encoding Other Activation Functions:* We will now describe the encoding for more general activation functions including tanh and sigmoid functions. Unlike a ReLU unit, that is described by a two piecewise linear function, approximating these functions may require three or more linear “pieces”. Furthermore, we would like our approximation to include an error estimate that bounds away the differences between the original function and its piecewise approximation.

We will encode the constraint  $y = \sigma(x)$  for a single neuron with  $x \in [-M_x, M_x]$ . The activation function  $y : \sigma(x)$  is approximated by a piecewise linear function:



Also, the output  $y$  is bounded inside the range  $[-M_y, M_y]$ . This bound is inferred by bounding  $\sigma(x)$  over inputs  $[-M_x, M_x]$ . The bound  $M_x$  is estimated conservatively for a given network through interval analysis.

To encode the constraint  $y = \sigma(x)$  as an MILP, we now introduce binary variables  $t_0, \dots, t_k \in \{0, 1\}^{k+1}$ , wherein  $t_i = 1$  encodes the case when  $x_i \leq x \leq x_{i+1}$ . For convenience, set  $x_0 = -M_x$  and  $x_{k+1} = M_x$ .

We encode that at most one of the cases can apply for any given  $x$ .

$$t_0 + \dots + t_k = 1$$

Next,  $t_i = 1$  must imply  $x_i \leq x \leq x_{i+1}$ :

$$x_i - 2(1 - t_i)M_x \leq x \leq x_{i+1} + 2(1 - t_i)M_x$$

Thus, if  $t_i = 1$  then the bounds are simply  $x_i \leq x \leq x_{i+1}$ . For  $t_i = 0$ , we get  $x_i - 2M_x \leq x \leq x_{i+1} + 2M_x$ , which follows from  $-M_x \leq x \leq M_x$ .

The output  $y$  is related to the inputs as

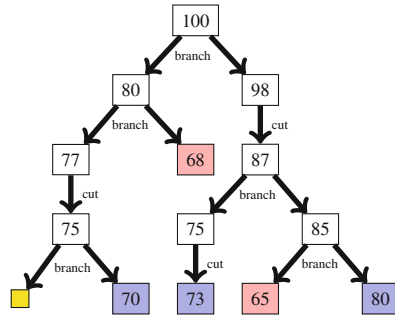
$$a_i x + b_i - \varepsilon_i - 2(1 - t_i)M_y \leq y \leq a_i x + b_i + \varepsilon_i + 2(1 - t_i)M_y.$$

Given the encoding for a single unit, we can now write down constraints for encoding an entire neural network with these activation units as an MILP, as shown earlier for ReLU units.

*Solving Monolithic MILP (Branch-and-Cut Algorithm):* Once the MILP is formulated, the overall problem can be handed off to a generic MILP solver, which yields an optimal solution. Most high performance MILP solvers are based on a branch-and-cut approach that will be briefly described here [20].

First, the approach solves the LP relaxation of the problem in (3.1) by temporarily treating the binary variables  $\mathbf{t}_1, \dots, \mathbf{t}_k$  as real-valued. The optimal solution of the relaxation is an upper bound to that of the original MILP. The two solutions are equal if the LP solver yields binary values for  $\mathbf{t}_1, \dots, \mathbf{t}_k$ . However, failing this, the algorithm has two choices to *eliminate* the invalid fractional solution:

- (a) Choose a fractional variable  $\mathbf{t}_{i,j}$ , and branch into two subproblems by adding the constraint  $\mathbf{t}_{i,j} = 0$  to one problem, and  $\mathbf{t}_{i,j} = 1$  to the other.
- (b) Add some valid inequalities (cutting planes) that remove the current fractional solutions but preserve all integer solutions to the problem.



**Fig. 2.** A tree representation of a branch-and-cut solver execution: each node shows the optimal value of the LP relaxation, if feasible. The leaves are color coded as yellow: infeasible, blue: feasible solution for MILP found, red: suboptimal to already discovered feasible solution. (Color figure online)

In effect, the overall execution of a branch-and-cut solver resembles a tree. Each node represents an MILP instance with the root being the original instance. Figure 2 depicts such a tree visually providing some representative values for the solution of the LP relaxation at each node. The leaves may represent many possibilities:

1. The LP relaxation yields an integral solution. In this case, we use this solution to potentially update the best solution encountered thus far, denoted  $z_{\max}$ . The leaves colored blue in Fig. 2 depict such nodes.
2. The LP relaxation is infeasible, eg., the yellow leaf in Fig. 2.
3. The LP relaxation’s objective is less than or equal to  $z_{\max}$ , the best feasible solution seen thus far. The leaves colored red in Fig. 2 depict this possibility.

Therefore, the key to solving MILPs fast lies in discovering feasible MILP solutions, early on, whose objective function values are as large as possible. In the subsequent section, we will describe how a local search procedure can be used to improve feasible solutions found by the solver (blue leaves in Fig. 2).

## 4 Combining MILP Solvers with Local Search

In this section, we will describe how local search can be used alongside an MILP solver to yield a more efficient solver for the range estimation problem. The key idea is to use local search on a connected subspace of the search space to improve any solution found by the global non-convex optimizer (MILP in our case).

### 4.1 Overall Approach

The overall approach is shown in Algorithm 1. It consists of two major components: A *local search* represented by the call to **LocalSearch** in line 7 and the call to **SolveMILPUptoThreshold** in line 9.

Local search uses gradient ascent over the neural network, starting from the current input  $\mathbf{x} \in P$  with  $u : F_N(\mathbf{x})$  to yield a new input  $\hat{\mathbf{x}}$  with  $\hat{u} : F_N(\hat{\mathbf{x}})$ , such that  $\hat{u} \geq u$ .

The MILP solver works over the MILP encoding (3.1), formulated in line 4. However, instead of solving the entire problem in one shot, the solver is provided a *target threshold*  $u$  as input. It searches for a feasible solution whose objective is at least  $u$ , and stops as soon as it finds one. Otherwise, it declares that no such solution is possible.

Packages such as Gurobi support such a functionality using the branch-and-cut solver by incrementally maintaining the current search tree. When called upon to find a solution that exceeds a given threshold, the solver performs sufficiently many steps of the branch-and-cut algorithm to either find a feasible solution that is above the requested threshold, or solve the problem to completion without finding such a solution. The approach shown in Algorithm 1 simply alternates between the local solver (line 7) and the MILP solver (line 9), while incrementing the current threshold by  $\delta$ , the tolerance parameter (line 8). We

---

**Algorithm 1.** Maximum value  $u$  for a neural network  $\mathcal{N}$  over  $\mathbf{x} \in P$  with tolerance  $\delta > 0$ .

---

```

1: procedure FINDUPPERBOUND( $\mathcal{N}, P, \delta$ )
2:    $\mathbf{x} \leftarrow \mathbf{Sample}(P)$  ▷ Sample an input at random
3:    $u \leftarrow \mathbf{EvalNetwork}(\mathcal{N}, \mathbf{x})$ 
4:    $I \leftarrow \mathbf{FormulateMILPEncoding}(\mathcal{N}, P)$  ▷ See (3.1)
5:   terminate  $\leftarrow$  false
6:   while not terminate do
7:      $(\hat{\mathbf{x}}, \hat{u}) \leftarrow \mathbf{LocalSearch}(\mathcal{N}, \mathbf{x}, P)$  ▷ Note:  $\hat{u} = F_N(\hat{\mathbf{x}})$ 
8:      $u \leftarrow \hat{u} + \delta$ 
9:      $(\mathbf{x}', u', \text{feas}) \leftarrow \mathbf{SolveMILPUptoThreshold}(I, u)$  ▷ Note: If feasible then
        $u' = F_N(\mathbf{x}')$ .
10:    if feasible then
11:       $(\mathbf{x}, u) \leftarrow (\mathbf{x}', u')$ 
12:    else
13:      terminate  $\leftarrow$  true
14:  return  $u$  ▷ return the upper bound  $u$ .

```

---

assume that the procedures **LocalSearch** and **SolveMILPUptoThreshold** satisfy the following properties:

- **(P1)** Given  $\mathbf{x} \in P$ , **LocalSearch** returns  $\hat{\mathbf{x}} \in P$  such that  $F_N(\hat{\mathbf{x}}) \geq F_N(\mathbf{x})$ .
- **(P2)** Given the encoding  $I$  and the threshold  $u$ , the **SolveMILPUptoThreshold** procedure either declares **feasible** along with  $\mathbf{x}' \in P$  such that  $u' = F_N(\mathbf{x}') \geq u$ , or declares **not feasible** if no  $\mathbf{x}' \in P$  satisfies  $F_N(\mathbf{x}') \geq u$ .

We recall the basic assumptions thus far: (a)  $P$  is compact, (b)  $\delta > 0$  and (c) properties **P1**, **P2** apply to the **LocalSearch** and **SolveMILPUptoThreshold** procedures. Let us denote the ideal upper bound by  $u^* : \max_{\mathbf{x} \in P} F_N(\mathbf{x})$ .

**Theorem 2.** *Algorithm 1 always terminates. Furthermore, the output  $u$  satisfies  $u \geq u^*$  and  $u \leq u^* + \delta$ .*

*Proof.* Since  $P$  is compact and  $F_N$  is a continuous function. Therefore, the maximum  $u^*$  is always attained for some  $\mathbf{x}^* \in P$ .

The value of  $u$  increases by at least  $\delta$  each time we execute the loop body of the While loop in line 6. Furthermore, letting  $u_0$  be the value of  $u$  attained by the sample obtained in line 2, we can upper bound the number of steps by  $\left\lceil \frac{(u^* - u_0)}{\delta} \right\rceil$ . This proves termination.

We note that the procedure terminates only if **SolveMILPUptoThreshold** returns infeasible. Therefore, appealing to property **P2**, we note that  $(\forall \mathbf{x} \in P) F_N(\mathbf{x}) \leq u$ . Or in other words,  $u^* \leq u$ .

Let  $u_n$  denote the value  $\hat{u}$  returned by **LocalSearch** in the final iteration of the loop and let the corresponding input be  $\mathbf{x}_n$ , so that  $F_N(\mathbf{x}_n) = u_n$ . We have  $u_n \leq u^* \leq u$ . However,  $u_n = u - \delta$ . Therefore,  $u \leq u^* + \delta$ .

## 4.2 Local Search Improvement

The local search uses a gradient ascent algorithm, starting from an input point  $\mathbf{x}_0 \in P$  and  $u = F_N(\mathbf{x}_0)$ , iterating through a sequence of points  $(\mathbf{x}_0, u_0), \dots, (\mathbf{x}_n, u_n)$ , such that  $\mathbf{x}_i \in P$  and  $u_0 < \dots < u_n$ . The new iterate  $\mathbf{x}_{i+1}$  is obtained from  $\mathbf{x}_i$ , in general, as follows:

1. Compute the gradient  $\mathbf{p}_i : \nabla F_N(\mathbf{x}_i)$ .
2. Find a new point  $\mathbf{x}_{i+1} := \mathbf{x}_i + s_i \mathbf{p}_i$  for a step size  $s_i > 0$ .

*Gradient Calculation:* Technically, the gradient of  $F_N(\mathbf{x})$  need not exist for each input  $\mathbf{x}$  if  $\sigma$  is a ReLU function. However, this happens for a set of points of measure 0, and is dealt with by using a smoothed version of the function  $\sigma$  defining the ReLU units.

The computation of the gradient uses the chain rule to obtain the gradient as a product of matrices:  $\mathbf{p} : J_0 \times J_1 \times \dots \times J_k$ , wherein  $J_i$  represents the Jacobian matrix of partial derivatives of the output of the  $(i+1)^{th}$  layer  $\mathbf{z}_{i+1}$  with respect to those of the  $i^{th}$  layer  $\mathbf{z}_i$ . Since  $\mathbf{z}_{i+1} = \sigma(W_i \mathbf{z}_i + \mathbf{b}_i)$  we can compute the gradient  $J_i$  using the chain rule. In practice, the gradient calculation can be piggybacked with function evaluation  $F_N(\mathbf{x})$  so that function evaluation returns both the output  $u$  and the gradient  $\nabla F_N(\mathbf{x})$ .

*Step Size Calculation:* First order optimization approaches present numerous rules such as the Armijo step sizing rules for calculating the step size [19]. Using these rules, we can use an off-the-shelf solver to compute a local maximum of  $F_N$ .

*Locally Active Regions:* Rather than perform steps using a step-sizing rule, we can perform longer steps for the special case of piecewise linear activation functions by defining a locally active region for the input  $\mathbf{x}$ . For the remainder of the discussion, we assume that  $\sigma$  is a piecewise linear function.

We first describe the concept for a ReLU unit. A ReLU unit is *active* if its input  $x \geq 0$ , and inactive otherwise.

**Definition 4 (Locally Active Region (ReLU)).** *For an input  $\mathbf{x}$  to the neural network  $\mathcal{N}$ , the locally active region  $\mathcal{L}(\mathbf{x})$  describes the set of all inputs  $\mathbf{x}'$  such that  $\mathbf{x}'$  activates exactly the same ReLU units as  $\mathbf{x}$ .*

The concept of locally active region can be generalized to any piecewise linear function  $\sigma$  that has  $J > 0$  pieces, say  $X_1, \dots, X_J$ , where  $\bigcup_i X_i$  is the input space and  $\sigma$  is linear on the input subspace  $X_i$ . For such a piecewise linear function  $\sigma$ , the *locally active region corresponding to an input  $\mathbf{x}$* ,  $\mathcal{L}(\mathbf{x})$ , is the set  $X_i$  such that  $\mathbf{x} \in X_i$ .

Given the definition of a locally active region, we obtain the following property for piecewise linear activation functions.

**Lemma 1.** *For all  $\mathbf{x}' \in \mathcal{L}(\mathbf{x})$ , we have  $\nabla F_N(\mathbf{x}) = \nabla F_N(\mathbf{x}')$ . Furthermore, for a ReLU neural net, the region  $\mathcal{L}(\mathbf{x})$  is described by a polyhedron with possibly strict inequality constraints.*

Let  $\overline{\mathcal{L}}(\mathbf{x}_i)$  denote the closure of the local active set obtained by converting the strict constraints (with  $>$ ) to their non-strict versions (with  $\geq$ ). Therefore, the local maximum is simply obtained by solving the following LP.

$$\max \mathbf{p}_i^T \mathbf{x} \text{ s.t. } \mathbf{x} \in \overline{\mathcal{L}}(\mathbf{x}_i) \cap P, \text{ where } \mathbf{p}_i : \nabla F_N(\mathbf{x}_i).$$

The solution of the LP above ( $\mathbf{x}_{i+1}$ ) yields the next iterate for local search. Note that this solution  $\mathbf{x}_{i+1}$  will typically be at the boundary of  $\overline{\mathcal{L}}(\mathbf{x}_i)$ . We randomly perturb this solution (or small numerical errors in the solver achieve the same effect) so that  $\nabla F_N(\mathbf{x}_{i+1}) \neq \nabla F_N(\mathbf{x}_i)$ .

*Termination:* The local search is terminated when each step no longer provides a sufficient increase, or alternatively the length of each step is deemed too small. These are controlled by user specified thresholds in practice. Another termination criterion simply stops the local search when a preset maximum number of iterations is exceeded. In our implementation, all three criteria are used.

**Lemma 2.** *Given a starting input  $\mathbf{x}_0 \in P$ , the **LocalSearch** procedure returns a new  $\mathbf{x}' \in P$ , such that  $F_N(\mathbf{x}') \geq F_N(\mathbf{x})$ .*

*Analysis:* For a given neural network  $\mathcal{N}$  and its corresponding MILP instance  $I$ , let us compare the number of nodes  $N_1$  explored by a monolithic solution to the MILP instance with the total number of nodes  $N_2$  explored collectively by the calls to the **SolveMILPUptoThreshold** routine in Algorithm 1. Additionally, let  $K_l$  denote the number of **LocalSearch** calls made by this algorithm. We expect each call to the local search to provide an improved feasible solution to the MILP solver, enabling it to potentially prune more nodes during its search. Therefore, the addition of local search is advantageous whenever

$$N_1 T_{lp} > N_2 T_{lp} + K_l T_{loc},$$

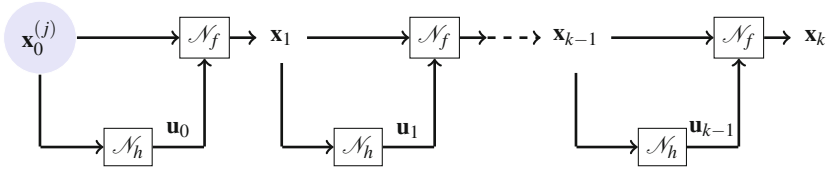
wherein  $T_{lp}$  is the average time taken to solve an LP relaxation (assumed to be the same) and  $T_{loc}$  is the average time for a local search.

A precise analysis is complicated since the future heuristic choices made by the solver can be different, due to the newly added local search iteration. Thus, we resort to an empirical comparison of the original monolithic MILP versus the solution procedure that uses the local search iterations.

## 5 Application: Reachability Analysis

Neural networks are increasingly used as models of physical dynamics and feedback control laws to achieve objectives such as safety, reachability and stability [13]. However, doing so in a verified manner is a challenging problem. We illustrate the computation reachable set over-approximations for such systems over a finite time horizon in order to prove bounded time temporal properties.



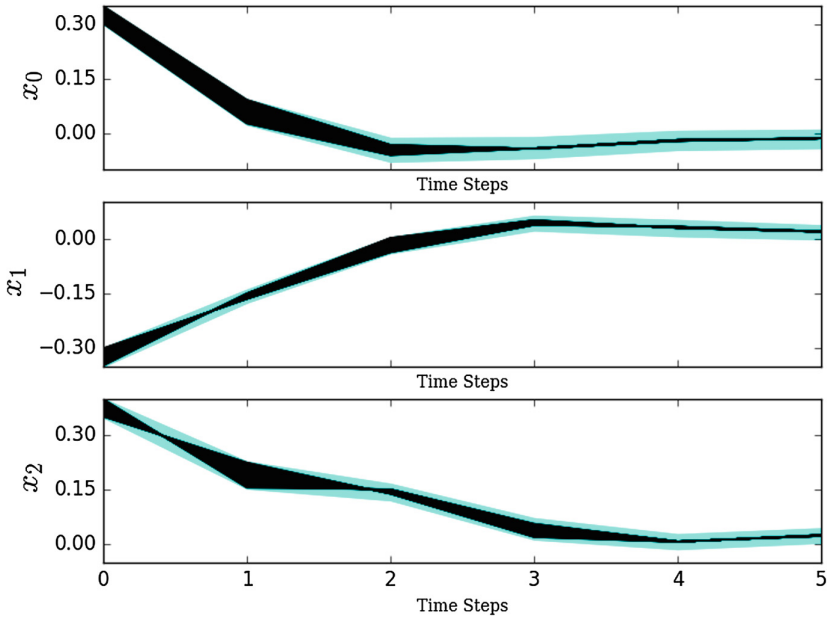


**Fig. 3.** Unwinding of the closed loop model with plant  $\mathcal{N}_f$  and controller  $\mathcal{N}_h$ , used to estimate reachable sets.

Figure 3 shows the unwinding of the plant network  $\mathcal{N}_f$  and the controller  $\mathcal{N}_h$ , for  $N > 0$  steps. Estimating an over-approximation of the reachable state  $\mathbf{x}_N$  at time  $N$  reduces to solving an output range analysis problem over the unwound network.

We trained a NN to control a nonlinear plant model (Example 17 from [24]) whose dynamics are describe by the ODE:  $\dot{x} = -x^3 + y$ ,  $\dot{y} = y^3 + z$ ,  $\dot{z} = u$ . We approximated the discrete time dynamics of the non linear system using a 4 input, 3 output neural network with 1 hidden layers having 300 neurons. This approximation ensures that unwinding, as shown in Fig. 3, results in a neural network.

Next, we devise a model predictive control (MPC) scheme to stabilize this system to the origin, and train the NN by sampling inputs from the state space



**Fig. 4.** Evolution of reach sets for the neural network feedback system.

$X : [-0.5, 0.5]^3$  and using the MPC to provide the corresponding control. We trained a 3 input, 1 output network, with 5 hidden layers, with the first layer having 100 neurons and the remaining 4 layers to saturate out the control output range.

For illustrative purpose, we compute the reach sets starting from the initial set,  $[0.3, 0.35] \times [-0.35, -0.3] \times [0.35, 0.4]$ , and compute the evolution as shown in Fig. 4. Note that, we stop the computation of reach sets once the sets are contained within the target box given by:  $[-0.05, 0.05] \times [-0.05, 0.05] \times [-0.05, 0.05]$ . The reach sets have been superimposed on numerous concrete system trajectories.

## 6 Experimental Evaluation

We have implemented the ideas described thus far in a C++-based tool called SHERLOCK. SHERLOCK combines local search with the commercial parallel MILP solver Gurobi, freely available for academic use [9]. Currently, our implementation supports neural networks with ReLU units. We hope to extend this to other activation functions, as described in the paper.

An interval analysis was used to set the  $M$  parameter in the MILP encoding. The tolerance parameter  $\delta$  in Algorithm 1, was set to  $5 \times 10^{-2}$  for all the test cases.

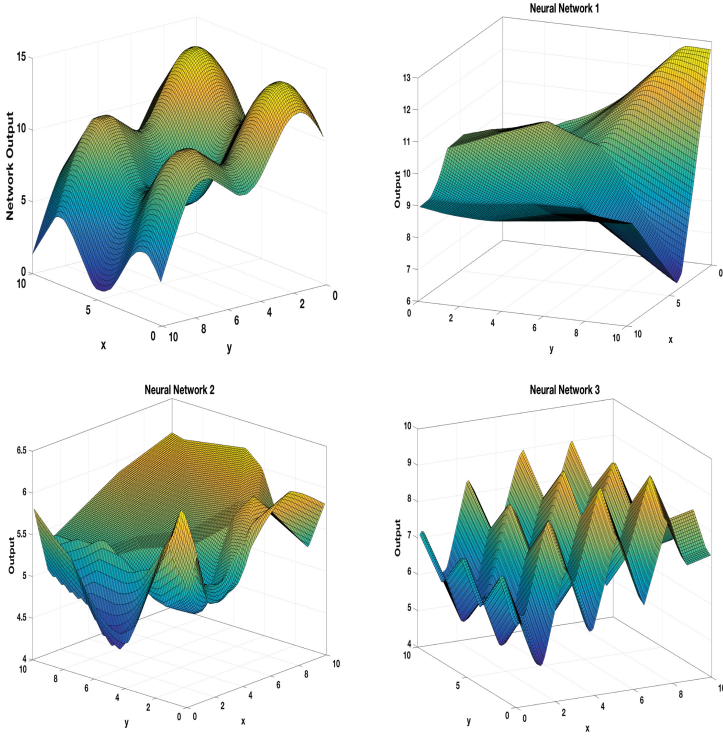
For comparison with Reluplex, we used the implementation available online [15]. However, at its core, Reluplex solves a satisfiability problem that checks if the output  $y$  lies inside a given range for constraints  $P$  over the inputs to the network. To facilitate comparisons, we simply use Reluplex to check if the output range computed by our approach is a valid over-approximation.

We consider a set of 16 microbenchmarks that consist of neural networks obtained from two different sources discussed below.

1. *Known Analytical Functions:* We formulated four simple analytical functions  $y = f(\mathbf{x})$  as shown in Fig. 5 controlling for the number of local minima seen over the chosen input range for each function. We then trained a neural network model based on input output samples  $(\mathbf{x}_i, f(\mathbf{x}_i))_{i=1}^N$  for each function. The result yielded networks  $N_0$ – $N_4$  along with the input constraints.
2. *Unwindings of closed loop systems:* We formulated 12 examples that come from the “unwinding” of a closed loop controller and plant models. The plant models are obtained from our previous work on controller synthesis [24]. The process of training the controller network is discussed elsewhere [7].

Table 1 summarizes the comparison of SHERLOCK against a “monolithic” MILP approach and the Reluplex solver. Since gurobi supports parallel branch-and-bound as a default, we report on the comparison over multicore (23 parallel cores) as well as single core deployments. Note that the comparison uses the total CPU time rather than wall clock time.

SHERLOCK on the multicore deployment is faster on all save one of the benchmarks in terms of CPU time. However, comparing the number of nodes explored,



**Fig. 5.** Plots of the first 4 benchmark functions used to train the neural networks shown in Table 1.

we observe that SHERLOCK explores fewer nodes in just 7 out of the 16 cases. We attribute this to the more complex nature of parallel branch-and-cut heuristics, wherein parallel threads may explore more nodes than strictly necessary. For the single core deployment, we note that the total CPU time is strongly correlated with the number of nodes explored. Here, SHERLOCK outperforms the monolithic MILP on the six largest examples with over 1000 neurons ( $N_{10}$ – $N_{15}$ ) in terms of time and number of nodes explored. For the smaller examples, the monolithic solver outperforms our approach, but the running times remain small for both approaches. For two of the networks, ( $N_7$  and  $N_{15}$ ), our starting sample followed by a local search resulted in the global maximum, which was certified by the LP relaxation. This leads to a node count of 1.

Comparing with Reluplex, we note that Reluplex was able to verify the bound in 6 instances but at a larger time cost than SHERLOCK or the monolithic MILP approach. For 10 out of 16 instances, the solver terminates due to an internal error.

**Table 1.** Performance results on networks trained on functions with known maxima and minima. **Legend:**  $x$  number of inputs,  $k$  number of layers,  $N$ : total number of neurons,  $T$ : CPU time taken,  $Nc$ : number of nodes explored. All the tests were run on a Linux server running Ubuntu 17.04 with 24 cores, and 64 GB RAM (DNC: Did Not Complete)

ID	$x$	$k$	$N$	23 cores				single core				Reluplex $T$
				SHERLOCK		Monolithic		SHERLOCK		Monolithic		
				$T$	$Nc$	$T$	$Nc$	$T$	$Nc$	$T$	$Nc$	
$N_0$	2	1	100	1s	94	2.3s	24	0.4s	44	0.3s	25	9.0
$N_1$	2	1	200	2.2s	166	3.6s	29	0.9s	71	0.8s	38	1m50s
$N_2$	2	1	500	7.8s	961	12.6s	236	2s	138	2.9s	257	15m59s
$N_3$	2	1	500	1.5s	189	0.5s	43	0.6s	95	0.2s	53	12m25s
$N_4$	2	1	1000	3m52s	32E3	3m52s	3E3	1m20s	4.8E3	35.6s	5.3E3	1h06m
$N_5$	3	7	425	4s	6	6.1s	2	1.7s	2	0.9s	2	DNC
$N_6$	3	4	762	3m47s	3.3E3	4m41s	3.6E3	37.8s	685	56.4s	2.2E3	DNC
$N_7$	4	7	731	3.7s	1	7.7s	2	3.9s	1	3.1s	2	1h35m
$N_8$	3	8	478	6.5s	3	40.8s	2	3.6s	3	3.3s	2	DNC
$N_9$	3	8	778	18.3s	114	1m11s	2	12.5s	12	4.3s	73	DNC
$N_{10}$	3	26	2340	50m18s	4.6E4	1h26m	6E4	17m12s	2.4E4	18m58s	1.9E4	DNC
$N_{11}$	3	9	1527	5m44s	450	55m12s	6.4E3	56.4s	483	130.7s	560	DNC
$N_{12}$	3	14	2292	24m17s	1.8E3	3h46m	2.4E4	8m11s	2.3E3	1h01m	1.6E4	DNC
$N_{13}$	3	19	3057	4h10m	2.2E4	61h08m	6.6E4	1h7m	1.5E4	15h1m	1.5E5	DNC
$N_{14}$	3	24	3822	72h39m	8.4E4	111h35m	1.1E5	5h57m	3E4	timeout	-	DNC
$N_{15}$	3	127	6845	2m51s	1	timeout	-	3m27s	1	timeout	-	DNC

## 7 Conclusion

We presented a combination of local and global search for estimating the output ranges of neural networks given constraints on the input. Our approach has been implemented inside the tool SHERLOCK and we compared our results with those obtained using the solver Reluplex. We also demonstrated the application of our approach to verification of NN-based control systems. Our approach can potentially be applied to verify controllers learned by reinforcement learning techniques.

Our main insight here is to supplement search over a nonconvex space by using local search over known convex subspaces. This idea is generally applicable. In this paper, we showed how this idea can be applied to range estimation of neural networks. The convex subspaces are obtained by fixing the subset of active neurons.

In the future, we wish to improve SHERLOCK in many directions, including the treatment of recurrent neural networks, handling activation functions beyond ReLU units and providing faster alternatives to MILP for global search.

**Acknowledgments.** We gratefully acknowledge inputs from Sergio Mover and Marco Gario for their helpful comments on an earlier version of this paper. This work was funded in part by the US National Science Foundation (NSF) under award numbers CNS-1646556, CNS-1750009, CNS-1740079 and US ARL Cooperative Agreement W911NF-17-2-0196. All opinions expressed are those of the authors and not necessarily of the US NSF or ARL.

## References

1. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
2. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. *Handb. Satisfiability* **185**, 825–885 (2009)
3. Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A., Criminisi, A.: Measuring neural net robustness with constraints. In: *Advances in Neural Information Processing Systems*, pp. 2613–2621 (2016)
4. Bixby, R.E.: A brief history of linear and mixed-integer programming computation. *Documenta Mathematica* 107–121 (2012)
5. Bunel, R., Turkaslan, I., Torr, P.H.S., Kohli, P., Kumar, M.P.: Piecewise linear neural network verification: a comparative study. CoRR, abs/1711.00455 (2017)
6. Dutta, S.: SHERLOCK: an output range analysis tool for neural networks. <https://github.com/souradeep-111/sherlock>
7. Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Verified inference of feedback control systems using feedforward neural networks. Draft (2017). Available upon request
8. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: D’Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 269–286. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-68167-2\\_19](https://doi.org/10.1007/978-3-319-68167-2_19)
9. Gurobi Optimization, Inc.: Gurobi optimizer reference manual (2016)
10. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. CoRR, abs/1610.06940 (2016)
11. IBM ILOG Inc.: CPLEX MILP Solver (1992)
12. Julian, K., Kochenderfer, M.J.: Neural network guidance for UAVs. In: *AIAA Guidance Navigation and Control Conference (GNC)* (2017)
13. Kahn, G., Zhang, T., Levine, S., Abbeel, P.: Plato: policy learning using adaptive trajectory optimization. arXiv preprint [arXiv:1603.00622](https://arxiv.org/abs/1603.00622) (2016)
14. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-63387-9\\_5](https://doi.org/10.1007/978-3-319-63387-9_5)
15. Katz, et al.: Reluplex: CAV 2017 prototype (2017). <https://github.com/guykatzz/ReluplexCav2017>
16. Kurd, Z., Kelly, T.: Establishing safety criteria for artificial neural networks. In: Palade, V., Howlett, R.J., Jain, L. (eds.) KES 2003. LNCS (LNAI), vol. 2773, pp. 163–169. Springer, Heidelberg (2003). [https://doi.org/10.1007/978-3-540-45224-9\\_24](https://doi.org/10.1007/978-3-540-45224-9_24)

17. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. *Nature* **521**(7553), 436–444 (2015)
18. Lomuscio, A., Maganti, L.: An approach to reachability analysis for feed-forward relu neural networks. CoRR, abs/1706.07351 (2017)
19. Luenberger, D.G.: *Optimization by Vector Space Methods*. Wiley, Hoboken (1969)
20. Mitchell, J.E.: Branch-and-cut algorithms for combinatorial optimization problems. In: *Handbook of Applied Optimization*, pp. 65–77 (2002)
21. Papernot, N., McDaniel, P.D., Goodfellow, I.J., Jha, S., Celik, Z.B., Swami, A.: Practical black-box attacks against deep learning systems using adversarial examples. CoRR, abs/1602.02697 (2016)
22. Pulina, L., Tacchella, A.: An abstraction-refinement approach to verification of artificial neural networks. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 243–257. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-14295-6\\_24](https://doi.org/10.1007/978-3-642-14295-6_24)
23. Pulina, L., Tacchella, A.: Challenging SMT solvers to verify neural networks. *AI Commun.* **25**(2), 117–135 (2012)
24. Sassi, M.A.B., Bartocci, E., Sankaranarayanan, S.: A linear programming-based iterative approach to stabilizing polynomial dynamics. In: *Proceedings of IFAC 2017*. Elsevier, Amsterdam (2017)
25. Scheibler, K., Winterer, L., Wimmer, R., Becker, B.: Towards verification of artificial neural networks. In: *MBMV Workshop*, pp. 30–40 (2015)
26. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R.: Intriguing properties of neural networks. CoRR, abs/1312.6199 (2013)
27. Tjeng, V., Tedrake, R.: Verifying neural networks with mixed integer programming. CoRR, abs/1711.07356 (2017)
28. Vanderbei, R.J.: *Linear Programming: Foundations & Extensions*, Second edn. Springer, Heidelberg (2001). Cf. <http://www.princeton.edu/~rvdb/LPbook/>
29. Williams, H.P.: *Model Building in Mathematical Programming*, 5th edn. Wiley, Hoboken (2013)
30. Xiang, W., Tran, H.-D., Johnson, T.T.: Output reachable set estimation and verification for multi-layer neural networks. CoRR, abs/1708.03322 (2017)
31. Xiang, W., Tran, H.-D., Rosenfeld, J.A., Johnson, T.T.: Reachable set estimation and verification for a class of piecewise linear systems with neural network controllers (2018). To Appear in the American Control Conference (ACC), Invited Session on Formal Methods in Controller Synthesis



# Formal Dynamic Fault Trees Analysis Using an Integration of Theorem Proving and Model Checking

Yassmeen Elderhalli<sup>(✉)</sup> , Osman Hasan , Waqar Ahmad ,  
and Sofiène Tahar 

Electrical and Computer Engineering, Concordia University, Montréal, Canada  
{y\_elderh, o\_hasan, waqar, tahar}@ece.concordia.ca

**Abstract.** Dynamic fault trees (DFTs) have emerged as an important tool for capturing the dynamic behavior of system failure. These DFTs are analyzed qualitatively and quantitatively using stochastic or algebraic methods. Model checking has been proposed to conduct the failure analysis of systems using DFTs. However, it has not been used for DFT qualitative analysis. Moreover, its analysis time grows exponentially with the number of states and its reduction algorithms are usually not formally verified. To overcome these limitations, we propose a methodology to perform the formal analysis of DFTs using an integration of theorem proving and model checking. We formalize the DFT gates in higher-order logic and formally verify many algebraic simplification properties using theorem proving. Based on this, we prove the equivalence between raw DFTs and their reduced forms to enable the formal qualitative analysis (determine the cut sets and sequences) of DFTs with theorem proving. We then use model checking to perform the quantitative analysis (compute probabilities of failure) of the formally verified reduced DFT. We applied our methodology on five benchmarks and the results show that the formally verified reduced DFT was analyzed using model checking with up to six times less states and up to 133000 times faster.

**Keywords:** Dynamic fault trees · Theorem proving · Model checking  
HOL4 · STORM

## 1 Introduction

A Fault Tree (FT) [1] is a graphical representation of the causes of failure of a system that is usually represented as the top event of the fault tree. FTs can be categorized as Static Fault Trees (SFT) and Dynamic Fault Trees (DFT) [1]. In an SFT, the structure function (expression) of the top event describes the failure relationship between the basic events of the tree using FT gates, like AND and OR, without considering the sequence of failure of these events. DFTs, on the other hand, model the failure behavior of the system using dynamic FT gates, like the spare gate, which can capture the dependent behavior of the basic events along with the static gates. DFTs model a more realistic representation

of systems compared to SFTs. For example, the spare DFT gate can model the failure of car tires and their spares, which cannot be modeled using SFT gates.

Fault Tree Analysis (FTA) [1] has become an essential part of the design process of safety-critical systems, where the causes of failure and their probabilities should be considered at an early stage. There are two main phases for FTA, the qualitative analysis and the quantitative analysis [2]. In the *qualitative analysis*, the cut sets and cut sequences are determined, which, respectively, represent combinations and sequences of basic events of the DFT that cause a system failure [1]. The *quantitative analysis* provides numeric results about the probability of failure of the top event and the mean-time-to-failure (MTTF) among other metrics [1]. Dynamic FTA is commonly carried out using algebraic [3] or Markov chain based approaches [2]. In the former, an algebra similar to the Boolean algebra is used to determine the structure function of the top event, which can be simplified to determine a reduced form of the cut sets and sequences. The probabilistic analysis of the DFT can then be performed based on the reduced form of the generated structure function by considering the probability of failure of the basic events. For the Markov chain based analysis, the FT is first converted to its equivalent Markov chain and then the probability of failure of the top event is determined by analyzing the generated Markov chain. The resultant Markov chain can be very large, while dealing with complex systems, which limits the usage of Markov chains in DFT analysis.

Traditionally, the dynamic FTA is performed using paper-and-pencil proof methods or computer simulation. While the former is error prone, specially for large systems, the latter provides a more scalable alternative. However, the results of simulation cannot be termed as accurate due to the involvement of several approximations in the underlying computation algorithms and the sampling based nature of this method. Given the dire need of accuracy in the failure analysis of safety-critical systems, formal methods have also been recently explored for DFT analysis. For example, the STORM probabilistic model checker [4] has been used to analyze DFTs based on Markov chain analysis [5]. However, probabilistic model checking has not been used in the formal qualitative analysis of DFTs. Moreover, it cannot support the analysis of large systems unless a reduction algorithm is invoked, and the implementation of such reduction is usually not formally verified. Therefore, one cannot ascertain that the analysis results after reduction are accurate or correspond to the original system. On the other hand, while in theory higher-order logic (HOL) theorem proving can cater for the above shortcomings, its support for FTA has been limited to SFTs [6].

In this paper, we propose to overcome the above-mentioned limitations of formal DFT analysis by using an integration of model checking and theorem proving. Firstly, we use theorem proving for modeling DFTs and verifying the equivalence between the original and the reduced form of the DFT. The formally verified reduced DFT can then be used for qualitative analysis (determining the cut sets and sequences) as well as for quantitative analysis using model checking. Thus, our proposed methodology tends to provide more sound results than sole model checking based analysis thanks to the involvement of a theorem



prover in the verification of the reduced model. Moreover, it caters for the state-space based issues of model checking by providing it with a reduced DFT model of a given system for the quantitative analysis. In order to illustrate the utilization and effectiveness of our proposed methodology, we analyzed five DFT benchmarks, namely: a Hypothetical Example Computer System (HECS) [2], a Hypothetical Cardiac Assist System (HCAS) [3, 7], a scaled cascaded PAND DFT [7, 8], a multiprocessor computing system [7, 9] and a variant of the Active Heat Rejection System (AHRS) [10]. The reduced DFTs and their reduced cut sequences are formally verified using the HOL4 theorem prover [11]. We use the STORM model checker to formally analyze the original as well as the reduced DFT. The results show that using the verified reduced DFT for the quantitative analysis allows us to reduce both the number of generated states by the model checker by up to 6 times and the time required to perform the analysis up to 133000 times faster.

## 2 Related Work

DFT analysis has been done using various tools and techniques [1]. For example, Markov chains have been extensively used for the modeling and analysis of DFTs [2]. The scalability of Markov chains in analyzing large DFTs is achieved by using a modularization approach [12], where the DFT is divided into two parts: static and dynamic. The static subtree is analyzed using ordinary SFT analysis methods, such as Binary Decision Diagrams (BDD) [1], and the dynamic subtree is analyzed using Markov chains. This kind of modularization approach is available in the Galileo tool [13]. In [7], the authors use a compositional aggregation technique to develop Input-Output Interactive Markov Chains (I/O-IMC) to analyse DFTs. This approach is implemented in the DFTCalc tool [14]. The algebraic approach has also been extensively used in the analysis of DFTs [3], where the top event of the DFT can be expressed and reduced in a manner similar to the ordinary Boolean algebra. The reliability of the system expressed algebraically can be evaluated based on the algebraic expression of the top event [8]. The main problem with the Markov chain analysis is the large generated state space when analyzing complex systems, which requires high resources in terms of memory and CPU time. Moreover, simulation is usually utilized in the analysis process, which does not provide accurate results. Although modularization tends to overcome the large state-space problem with Markov chains, we cannot obtain a verified reduced form of the cut sequences of the DFT. The algebraic approach provides a framework for performing both the reduction and the analysis of the DFT. However, the foundations of this approach have not been formalized, which implies that the results of the analysis should not be relied upon especially in safety-critical systems.

Formal methods can overcome the above-mentioned inaccuracy limitations of traditional DFT analysis techniques. Probabilistic model checkers, such as STORM [4], have been used for the analysis of DFTs. The main idea behind this approach is to automatically convert the DFT of a given system into its corre-

sponding Markovian model and then analyze the safety characteristics quantitatively of the given system using the model checker [15]. The STORM model checker accepts the DFT to be analyzed in the Galileo format [13] and generates a failure automata of the tree. This approach allows us to verify failure properties, like probability of failure, in an automatic manner. However, the approach suffers from scalability issues due to the inherent state-space explosion problem of model checking for large systems. Moreover, the implementation of the reduction algorithms used in model checkers are generally not formally verified. Finally, model checkers have only been used in the context of probabilistic analysis of DFTs and not for the qualitative analysis, as the cut sequences in the qualitative analysis cannot be provided unless the state machine is traversed to the fail state, which is difficult to achieve for large state machines.

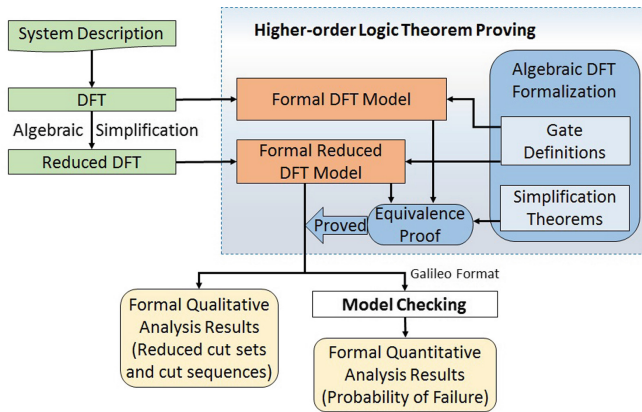
Exploiting the expressiveness of higher-order logic (HOL) and the soundness of theorem proving, Ahmad and Hasan [6, 16] formalized SFTs in HOL4 and evaluated the probability of failure based on the Probabilistic Inclusion-Exclusion principle. However, the main problem in theorem proving lies in the fact that it is interactive, i.e., it needs user guidance in the proof process. Moreover, to the best of our knowledge, no higher-order-logic formalization of DFTs is available in the literature so far and thus it is not a straightforward task to conduct the DFT analysis using a theorem prover as of now.

It can be noted that both model checking and HOL theorem proving exhibit complementary characteristics, i.e., model checking is automatic but cannot deal with large systems and does not provide qualitative analysis of DFTs, while HOL theorem proving allows us to verify universally quantified generic mathematical expressions but at the cost of user interventions. In this paper, we leverage upon the complementary nature of these approaches to present an integrated methodology that provides the expressiveness of higher-order logic and the existing support for automated probabilistic analysis of DFTs using model checking. The main idea is to use theorem proving to formally verify the equivalence between the original and the reduced DFT and then use a probabilistic model checker to conduct a quantitative analysis on the reduced DFT. As a result, a formally verified reduced form of the cut sequences is obtained. In addition, both the generated state machine and the analysis time are reduced.

### 3 Proposed Methodology

Our proposed methodology of the formal DFT analysis is depicted in Fig. 1. It provides both formal DFT qualitative analysis using theorem proving and quantitative analysis using model checking. The DFT analysis starts by having a system description. The failure behavior of this system is then modeled as a DFT, which can be reduced based on the algebraic approach [3]. The idea of this algebraic approach is to deal with the events, which can represent the basic events or outputs, according to their time of failure ( $d$ ). For example,  $d(X)$  represents the time of failure of an event  $X$ . In the algebraic approach, temporal operators (Simultaneous ( $\Delta$ ), Before ( $\triangleleft$ ) and Inclusive Before ( $\trianglelefteq$ ))

are defined to model the dynamic gates. Based on these temporal operators, several simplification theorems exist to perform the required reduction. This reduction process can be erroneous if it is performed manually using paper-and-pencil. Moreover, reduction algorithms may also provide wrong results if they are not formally verified. In order to formally check the equivalence between the original model and the reduced one, we developed a library of formalized dynamic gates in HOL and verified their corresponding simplification theorems [17]. These foundations allow us to develop a formal model for any DFT using the formal gate definitions. Based on the verified simplification theorems, we can then verify the equivalence between the formally specified original and the reduced DFT models using a theorem prover. The formally verified reduced DFT can then be utilized to perform the qualitative analysis of the reduced model in the theorem prover as well as its quantitative analysis by using a model checker.



**Fig. 1.** Overview of proposed methodology

The qualitative analysis represents an important and crucial step in DFT analysis, since it allows to identify the sources of failure of the system without the availability of any information or actual numbers about the failure probabilities of the basic events. In SFTs, the qualitative analysis is performed by finding the cut sets, which are the combination of events that causes system failure without providing any information about the required sequence that will cause the failure. The temporal behavior of dynamic gates allows representing the failure dependencies and sequences in a system. Due to this temporal behavior of the dynamic gates, just finding the cut sets does not capture the sequence of failure of events that can cause the system failure. The cut sequences on the other hand capture not only the combination of basic events but also the sequence of events that can cause the system failure. In the proposed methodology, we utilize a HOL theorem prover to verify a reduced expression of the structure function of the top event, which ensures that the reduction process is correct. Using this

reduced structure function, a formally verified reduced form of the cut sequences can also be determined.

The reduced form of the structure function of the top event can now be used in a probabilistic model checker to do the quantitative analysis of the given system. Because of the reduced model, we get a reduction in the analysis time and number of states. In this paper, the STORM model checker is used to perform the probabilistic analysis of the DFT. Several input languages are supported by this model checker including the Galileo format for DFT. Both the probability of failure of the top event as well as the mean time to failure can be computed using STORM. It is worth mentioning that since the analyzed model of the DFT is a Markov Automata (MA) (in case of non-deterministic behavior) or a Continuous Time Markov Chain (CTMC), only exponential failure distributions are supported by the proposed methodology.

## 4 Formalization of Dynamic Fault Trees in HOL

In this section, we present the formal definitions of the identity elements, the temporal operators and the dynamic gates. It is assumed that a fault is represented using an event. The occurrence of a fault indicates that the corresponding failure event is true. It is also assumed that the events are non-repairable.

### 4.1 Identity Elements

Two identity elements are defined, namely the *ALWAYS* and the *NEVER* elements. The *ALWAYS* identity element represents an event with a failure time equal to 0. The *NEVER* element represents an event that never occurs. These two elements are defined based on their time of failure in HOL as follows:

**Definition 1.** *ALWAYS element*

$\vdash \text{ALWAYS} = (0 : \text{extreal})$

**Definition 2.** *NEVER element*

$\vdash \text{NEVER} = \text{PosInf}$

where *extreal* is the HOL data-type for extended real numbers, which are real numbers including positive infinity ( $+\infty$ ) and negative infinity ( $-\infty$ ). The *PosInf* is a HOL symbol representing ( $+\infty$ ).

### 4.2 Temporal Operators

We also formalize three temporal operators in order to model the dynamic behavior of the DFT: *Simultaneous* ( $\Delta$ ), *Before* ( $\triangleleft$ ) and *Inclusive Before* ( $\trianglelefteq$ ). The *Simultaneous* operator has two input events, representing basic events or subtrees. The time of occurrence (failure) of the output event of this operator is

equal to the time of occurrence of the first or the second input event considering that both input events occur at the same time:

$$d(A\Delta B) = \begin{cases} d(A), & d(A) = d(B) \\ +\infty, & d(A) \neq d(B) \end{cases} \quad (1)$$

For any two basic events, if the failure distribution of the random variables associated with these basic events is continuous, then they cannot have the same time of failure [3], and hence the result of the *Simultaneous* operator between them is *NEVER*.

$$d(A\Delta B) = NEVER \quad (2)$$

where  $A$  and  $B$  are basic events with random variables that exhibit continuous failure distributions.

The *Before* operator accepts two input events, which can be basic events or two subtrees. The time of occurrence of the output event of this operator is equal to the time of occurrence of the first input event if the first input event (from left) occurs before the second input event (right), otherwise the output never fails:

$$d(A \triangleleft B) = \begin{cases} d(A), & d(A) < d(B) \\ +\infty, & d(A) \geq d(B) \end{cases} \quad (3)$$

The *Inclusive Before* combines the behavior of both the *Simultaneous* and *Before* operators, i.e., if the first input event (left) occurs before or at the same time as the second input event (right), then the output event occurs with a time equal to the time of occurrence of the first input event:

$$d(A \trianglelefteq B) = \begin{cases} d(A), & d(A) \leq d(B) \\ +\infty, & d(A) > d(B) \end{cases} \quad (4)$$

We formalize these temporal operators in HOL as follows:

**Definition 3.** *Simultaneous Operator*

$\vdash \forall (A : \text{extreal}) B. \text{D\_SIMULT } A \ B = \text{if } (A = B) \text{ then } A \text{ else PosInf}$

**Definition 4.** *Before Operator*

$\vdash \forall (A : \text{extreal}) B. \text{D\_BEFORE } A \ B = \text{if } (A < B) \text{ then } A \text{ else PosInf}$

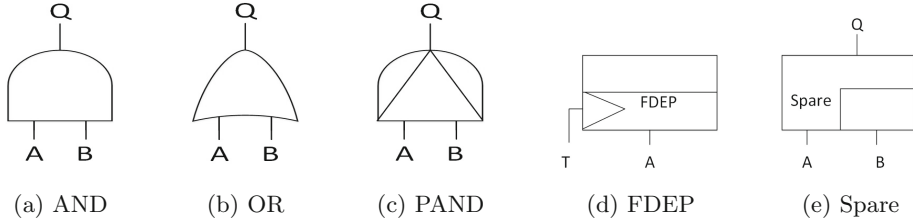
**Definition 5.** *Inclusive Before Operator*

$\vdash \forall (A : \text{extreal}) B. \text{D\_INCLUSIVE\_BEFORE } A \ B = \text{if } (A \leq B) \text{ then } A \text{ else PosInf}$

where  $A$  and  $B$  represent the time of failure of the events  $A$  and  $B$ , respectively.

### 4.3 Fault Tree Gates

Figure 2 shows the main FT gates [2]; dynamic gates as well as the static ones. Although, the AND ( $\cdot$ ) and OR ( $+$ ) gates, shown in Figs. 2a and b, are considered as static operators or gates, their behavior can be represented using the time of



**Fig. 2.** Fault tree gates

occurrence of the input events. For example, the output event of an AND gate occurs if and only if all its input events occur. This implies that the output of the AND gate occurs with the occurrence of the last input event, which means that the time of occurrence of the output event equals the maximum time of occurrence of the input events. The OR gate is defined in a similar manner with the only difference that the output event occurs with the occurrence of the first input event, i.e., the minimum time of occurrence of the inputs:

$$d(A \cdot B) = \max(d(A), d(B)) \tag{5}$$

$$d(A + B) = \min(d(A), d(B)) \tag{6}$$

We model the behavior of these gates in HOL as follows:

**Definition 6.** *AND gate (operator)*

$$\vdash \forall (A : \text{extreal}) B. D\_AND A B = \max A B$$

**Definition 7.** *OR gate (operator)*

$$\vdash \forall (A : \text{extreal}) B. D\_OR A B = \min A B$$

where `max` and `min` are HOL functions that return the maximum and the minimum values of their arguments, respectively.

The Priority-AND (PAND) gate is a special case of the AND gate, where the output occurs when all the input events occur in a sequence, conventionally from left to right. For the PAND gate, shown in Fig. 2c, the output  $Q$  occurs if  $A$  and  $B$  occur and  $A$  occurs before or with  $B$ . The behavior of the PAND gate can be represented using the time of failure as:

$$d(Q) = \begin{cases} d(B), & d(A) \leq d(B) \\ +\infty, & d(A) > d(B) \end{cases} \tag{7}$$

The behavior of the PAND can be expressed using the temporal operators as:

$$Q = B \cdot (A \preceq B) \tag{8}$$

We define the PAND gate in HOL as:

**Definition 8.** *PAND gate*

$$\vdash \forall (A : \text{extreal}) B. \text{PAND } A \ B = \text{if } (A \leq B) \text{ then } B \text{ else PosInf}$$

We verify in HOL that the PAND exhibits the behavior given in Eq. 8:

**Theorem 1.**  $\vdash \forall A \ B. \text{PAND } A \ B = \text{D\_AND } B \ (\text{D\_INCLUSIVE\_BEFORE } A \ B)$

The Functional DEpendency gate (FDEP), shown in Fig. 2d, is used when there is a failure dependency between the input events or sub-trees, i.e., the occurrence of one input (or a sub-tree) can trigger the occurrence of other input events (or subtrees) in the fault tree. For example, in Fig. 2d, the occurrence of  $T$  triggers the occurrence of  $A$ . This states that  $A$  occurs in two different ways: firstly, when  $A$  occurs by itself and secondly, when the trigger  $T$  occurs. This implies that the time of failure of  $A_T$  (triggered  $A$ ) equals the minimum time of occurrences of  $T$  and  $A$ :

$$d(A_T) = \min(d(A), d(T)) \quad (9)$$

We define the FDEP gate in HOL as:

**Definition 9.** *FDEP gate*

$$\vdash \forall (A : \text{extreal}) T. \text{FDEP } A \ T = \min A \ T$$

where  $T$  is the occurrence time of the trigger. We also verify in HOL that the FDEP gate is equivalent to an OR gate as follows:

**Theorem 2.**  $\vdash \forall A \ T. \text{FDEP } A \ T = \text{D\_OR } A \ T$

The spare gate, shown in Fig. 2e, represents a dynamic behavior that occurs in many real world systems, where we usually have a main part and some spare parts. The spare parts are utilized when the main part fails. The spare gate, shown in Fig. 2e, has a main input ( $A$ ) and a spare input ( $B$ ). After the failure of input  $A$ , the spare input  $B$  is activated. The output of the spare gate fails if both the main input and the spare fail. The spare gate can have several spare inputs, and the output fails after the failure of the main input and all the spares. The spare gate has three variants depending on the failure behavior of the spare part: hot spare gate (HSP), cold spare gate (CSP) or warm spare gate (WSP). In the HSP, the probability of failure for the spare is the same in both the dormant and the active states. For the CSP, the spare part cannot fail unless it is activated. The WSP is the general case, where the spare part can fail in the dormant state as well as in the active state, but the failure distribution of the spare in its dormant state is different from the one in the active mode, and it is usually attenuated by a dormancy factor. In order to be able to distinguish between the different states of the spare input, two different variables are assigned to each state. For example, for the spare gate, shown in Fig. 2e,  $B$  will be represented using two variables;  $B_d$  for the dormant state and  $B_a$  for the active state.

The input events of the spare gate cannot occur at the same time if they are basic events. However, if these events are subtrees then they can occur at the same time. For a two input warm spare gate, with  $A$  as the primary input and  $B$  as the spare input, the output event occurs in two ways; firstly, if  $B$  fails in its dormant state (inactive) then  $A$  fails with no spare to replace it. The second way is when  $A$  fails first then  $B$  (the spare part) is activated and then  $B$  fails in its active state. For the general case, when the input events can occur at the same time (if they are subtrees or depend on a common trigger), an additional option for the failure of the spare gate is added considering the two input events occur at the same time. The general form of the warm spare gate can be expressed mathematically as:

$$Q = A.(B_d \triangleleft A) + B_a.(A \triangleleft B_a) + A\Delta B_a + A\Delta B_d \quad (10)$$

We formally define the WSP in HOL as:

**Definition 10.** *Warm Spare Gate*

$\vdash \forall A B\_a B\_d.$

WSP A B\_a B\_d =

D\_OR(D\_OR (D\_OR (D\_AND A (D\_BEFORE B\_d A))

(D\_AND B\_a (D\_BEFORE A B\_a))) (D\_SIMULT A B\_a)) (D\_SIMULT A B\_d)

The time of failure of the CSP gate with primary input  $A$  and cold spare  $B$  can be defined as:

$$d(Q) = \begin{cases} d(B), & d(A) < d(B) \\ +\infty, & d(A) \geq d(B) \end{cases} \quad (11)$$

The above equation describes that the output event of the CSP occurs if the primary input fails and then the spare fails while in its active state. We define the CSP in HOL as:

**Definition 11.** *Cold Spare Gate*

$\vdash \forall (A : \text{extreal}) B. \text{CSP } A B = \text{if } (A < B) \text{ then } B \text{ else PosInf}$

We formally verify in HOL that the CSP gate is a special case of WSP, where the spare part cannot fail in its dormant state.

**Theorem 3.**  $\vdash \forall A B\_a B\_d.$

ALL\_DISTINCT [A; B\_a]  $\wedge$  COLD\_SPARE B\_d  $\Rightarrow$  (WSP A B\_a B\_d = CSP A B\_a)

where the predicate ALL\_DISTINCT ensures that  $A$  and  $B\_a$  are not equal, implying that they cannot fail at the same time, and COLD\_SPARE B\_d makes sure that the spare  $B$  is a cold spare, i.e., it cannot fail in its dormant mode ( $B\_d$ ).

The failure distribution of the spare part in the HSP remains the same in both states, i.e., the dormant and the active states. The output of the HSP fails when both the primary and the spare inputs fail, and the sequence of failures



does not matter, as the spare part has only one failure distribution. The HSP can be expressed mathematically as:

$$d(Q) = \max(d(A), d(B)) \quad (12)$$

where  $A$  is the primary input and  $B$  is the spare. It is formally defined in HOL as:

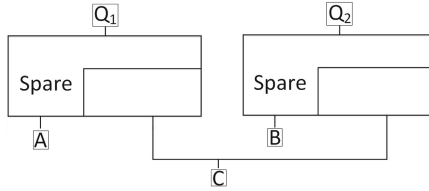
**Definition 12.** *Hot Spare Gate*

$\vdash \forall (A : \text{extreal}) B. \text{HSP } A \ B = \max A \ B$

We formally verify in HOL that if both the dormant and the active states of the spare are equal, then the WSP is equivalent to the HSP:

**Theorem 4.**  $\vdash \forall A \ B\_a \ B\_d. (B\_a = B\_d) \Rightarrow (\text{WSP } A \ B\_a \ B\_d = \text{HSP } A \ B\_a)$

It is important to mention that more than one spare gate can share the same spare input. In this case, there is a possibility that one of the primary inputs is replaced by the spare, while the other input does not have a spare in case it fails. The outputs of the spare gates, shown in Fig. 3, are expressed mathematically as follows (assuming that  $A$ ,  $B$  and  $C$  are basic events):



**Fig. 3.** Spare gates with shared spare

$$Q_1 = A.(C_d \triangleleft A) + C_a.(A \triangleleft C_a) + A.(B \triangleleft A) \quad (13)$$

$$Q_2 = B.(C_d \triangleleft B) + C_a.(B \triangleleft C_a) + B.(A \triangleleft B) \quad (14)$$

The last term in Eq. 13 ( $A.(B \triangleleft A)$ ) indicates that if  $B$  occurs before  $A$ , then the spare part  $C$  is used by the second spare gate. This implies that no spare is available for the first spare gate, which causes the failure of the output of the first spare gate if  $A$  occurs. We formalize the output  $Q_1$  of the first spare gate in HOL as:

**Definition 13.** *Shared Spare*

$\vdash \forall A \ B \ C\_a \ C\_d.$

`shared_spare A B C_a C_d =`

`D_OR (D_OR (D_AND A (D_BEFORE C_d A)) (D_AND C_a (D_BEFORE A C_a)))`  
`(D_AND A (D_BEFORE B A))`

#### 4.4 Formal Verification of the Simplification Theorems

As with classical Boolean algebra, many simplification theorems also exist for DFT operators, which can be used to simplify the structure function of the DFT [3]. We formally verified over 80 simplification theorems for the operators, defined in the previous subsection, including commutativity, associativity and idempotence of the AND, OR and Simultaneous operators, in addition to more complex theorems that include a combination of temporal operators. The verification process of these theorems was mainly based on the properties of extended real numbers, since the DFT operators are defined based on the time of failure of the events, defined using the `extreal` data-type in HOL. During the verification process, most sub-goals were automatically verified using tactics that utilize theorems from the `extreal` HOL theory. Some of these formally verified theorems are listed in Table 1. The complete list of formally verified theorems and more details about their verification can be found in [18].

**Table 1.** Some formally verified simplification theorems

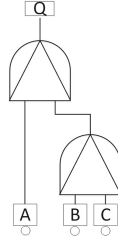
DFT Algebra Theorems	HOL Theorems
$A+B=B+A$	$\vdash \forall A B. D\_OR A B = D\_OR B A$
$A.B=B.A$	$\vdash \forall A B. D\_AND A B = D\_AND B A$
$A+A=A$	$\vdash \forall A. D\_OR A A = A$
$A.NEVER=NEVER$	$\vdash \forall A. D\_AND A NEVER = NEVER$
$A\Delta B=B\Delta A$	$\vdash \forall A B. D\_SIMULT A B = D\_SIMULT B A$
$(A\triangleleft B).(B\triangleleft A)=NEVER$	$\vdash \forall A B. D\_AND (D\_BEFORE A B)$ $(D\_BEFORE B A) = NEVER$
$A\triangleleft(B+C)=(A\triangleleft B).(A\triangleleft C)$	$\vdash \forall A B C. D\_BEFORE A (D\_OR B C) =$ $D\_AND (D\_BEFORE A B) (D\_BEFORE A C)$
$(A\triangleleft B).(B\triangleleft A)=A\Delta B$	$\vdash \forall A B. D\_AND (D\_INCLUSIVE\_BEFORE A B)$ $(D\_INCLUSIVE\_BEFORE B A) = D\_SIMULT A B$
$(A\triangleleft B)+(A\Delta B)=A\triangleleft B$	$\vdash \forall A B. D\_OR (D\_INCLUSIVE\_BEFORE A B)$ $(D\_SIMULT A B) = D\_INCLUSIVE\_BEFORE A B$
$(A\triangleleft B)+(A\Delta B)+(A.(B\triangleleft A))=A$	$\vdash \forall A B. D\_OR (D\_OR (D\_BEFORE A B)$ $(D\_SIMULT A B)) (D\_AND A D\_BEFORE B A)) = A$

Figure 4 shows an example of a simple DFT [8]. This DFT consists of two cascaded PAND gates with three basic events;  $A$ ,  $B$  and  $C$ . The temporal operators can be used to express the behavior of the PAND gate as follows [8]:

$$Q = C.(B \triangleleft C).(A \triangleleft (C.(B \triangleleft C))) \quad (15)$$

Using this expression, we cannot determine the required sequence of failure for the basic events that will cause the system failure, since the basic events are repeated in the expression. Using the algebraic simplification theorems, this structure function can be reduced to [8]:

$$Q = C.(B \triangleleft C).(A \triangleleft C) \quad (16)$$



**Fig. 4.** Simple DFT example

We verified this reduction using HOL4, which implies that this reduction process is correct and the new reduced DFT expression reflects the behavior of the original DFT.

$$\begin{aligned}
 &\vdash \forall A B C. \\
 &\quad \text{ALL\_DISTINCT } [A; B; C] \Rightarrow \\
 &\quad (\text{PAND } A \text{ (PAND } B \text{ } C) = \\
 &\quad \text{D\_AND } C \text{ (D\_AND (D\_BEFORE } A \text{ } C) \text{ (D\_BEFORE } B \text{ } C)))
 \end{aligned}$$

From this reduced expression, we can identify that two different sequences can cause the system failure;  $[A, B, C]$  or  $[B, A, C]$ .

$$Q = C . (A \triangleleft C) . (B \triangleleft A) + C . (B \triangleleft C) . (A \triangleleft B) \quad (17)$$

This reduced form of the structure function is verified in HOL to be equal to the top event of the original tree as:

$$\begin{aligned}
 &\vdash \forall A B C. \\
 &\quad \text{ALL\_DISTINCT } [A; B; C] \Rightarrow \\
 &\quad (\text{PAND } A \text{ (PAND } B \text{ } C) = \\
 &\quad \text{D\_OR}(\text{D\_AND } C \text{ (D\_AND (D\_BEFORE } A \text{ } C) \text{ (D\_BEFORE } B \text{ } A))) \\
 &\quad \text{(D\_AND } C \text{ (D\_AND (D\_BEFORE } B \text{ } C) \text{ (D\_BEFORE } A \text{ } B))))
 \end{aligned}$$

Since we have formally verified that the structure function is composed of the above-mentioned two sequences, we can conclude that the system will fail if any of these sequences occurs. In order to prevent or reduce the probability of failure of the top event, we should prevent the occurrence of these sequences, i.e., we should prevent the failure of  $A$  and  $B$  before  $C$ . This means that using the first part of our proposed methodology, we have been able to obtain a verified reduced form of the top event as well as a verified reduced form of the cut sequences.

## 5 Experimental Results

In order to illustrate the effectiveness of our proposed methodology, we conducted the formal DFT analysis of five benchmarks. The first benchmark, depicted in Fig. 5, is a scaled version of the original cascaded PAND DFT [7, 8] with repeated

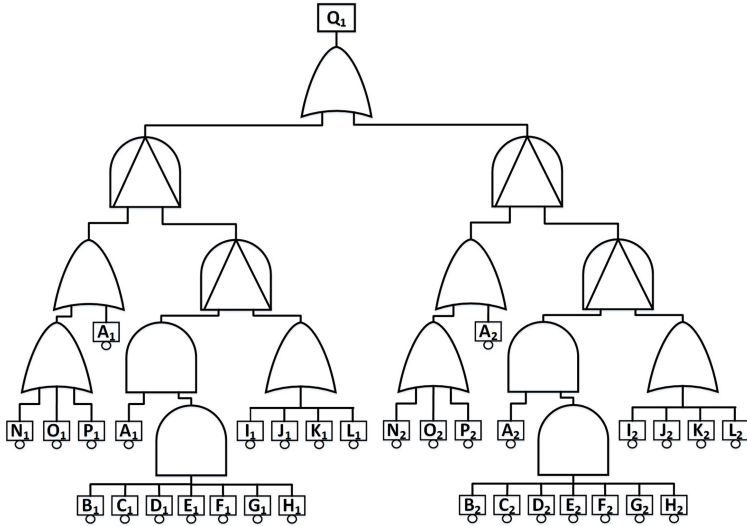


Fig. 5. Scaled cascaded PAND DFT

events. It has two similar subtrees with different basic events and a top event that fails whenever one of these subtrees fails. The second DFT is a modified and abstracted version of the Active Heat Rejection System (AHRS) [10], which consists of two thermal rejection units  $A$  and  $B$ . The failure of any of these two units leads to the failure of the whole system. Each main input ( $A$  or  $B$ ) has two spare parts, and the unit fails with the failure of the main input and the spare inputs. All the inputs are functionally dependent on the power supply. The third benchmark represents a Multiprocessor Computer System (MCS) [7, 9] with two redundant computers, having a processor, a disk and a memory unit. Each disk has its own spare and the two memory units share the same spare. The two processors are functionally dependent on the power supply. The fourth benchmark is a Hypothetical Example Computer System (HECS) [2] consisting of two processors with a cold spare, five memory units, which are functionally dependent on two memory interface units and two system buses. The failure of the system also depends on the application subsystem, which in turn depends on the software, the hardware and the human operator. The last benchmark is a Hypothetical Cardiac Assist System (HCAS) [3, 7], which consists of two bumps with a shared spare, two motors and a CPU with a spare. Both CPUs are functionally dependent on a trigger, which represents the crossbar switch and the system supervisor. In the sequel, we describe the formal analysis of the first benchmark. Details of the rest of the benchmarks as well as the HOL4 scripts and STORM models are available at [17, 18].

## 5.1 Formal Verification of the Reduced Cascaded PAND DFT

The first step in the proposed methodology is to create a formal model for both the original DFT and the reduced one. Then, the equivalence property between them is formally verified in HOL. This is followed by determining the cut sets and sequences. The top event ( $Q_1$ ) of the system, shown in Fig. 5, is reduced using the simplification theorems and this reduction is verified in HOL as follows [18]:

**Theorem 5.**  $\vdash \forall A_1 B_1 C_1 D_1 E_1 W_1 G_1 H_1 I_1 J_1 K_1 L_1 N_1 O_1 P_1 A_2 B_2 C_2 D_2 E_2$   
 $W_2 G_2 H_2 I_2 J_2 K_2 L_2 N_2 O_2 P_2. \text{ALL\_DISTINCT} [A_1; B_1; C_1; D_1; E_1; W_1; G_1; H_1;$   
 $I_1; J_1; K_1; L_1; N_1; O_1; P_1; A_2; B_2; C_2; D_2; E_2; W_2; G_2; H_2; I_2; J_2; K_2; L_2; N_2; O_2; P_2] \Rightarrow$   
 $(Q_1 = (I_1 + J_1 + K_1 + L_1) \cdot (A_1 \triangleleft (I_1 + J_1 + K_1 + L_1)) \cdot ((B_1 \cdot C_1 \cdot D_1 \cdot E_1 \cdot W_1 \cdot G_1 \cdot H_1) \triangleleft (I_1 + J_1 + K_1 + L_1))$   
 $+ (I_2 + J_2 + K_2 + L_2) \cdot (A_2 \triangleleft (I_2 + J_2 + K_2 + L_2)) \cdot ((B_2 \cdot C_2 \cdot D_2 \cdot E_2 \cdot W_2 \cdot G_2 \cdot H_2) \triangleleft (I_2 + J_2 + K_2 + L_2)))$

The predicate `ALL_DISTINCT` ensures that the basic events cannot occur at the same time. This condition was found to be a prerequisite for the above-mentioned consequence. We can observe from the above theorem that the basic events ( $N_1, O_1, P_1, N_2, O_2, P_2$ ) have no effect on the failure of the top event since they are eliminated in the reduction. Considering the cut sets and sequences, the top event can fail in two cases. The first case corresponds to the first product in the structure function, which implies that the output event occurs if any one of the basic events ( $I_1, J_1, K_1, L_1$ ) occurs and  $A_1$  occurs before all of them and the inputs ( $B_1, C_1, D_1, E_1, W_1, G_1, H_1$ ) occur before the inputs ( $I_1, J_1, K_1, L_1$ ). The second case represents the second product of the second subtree, which is similar to the first product but with different basic events. Since the Galileo format (which is used to model a DFT in STORM) supports only DFT gates and not operators, it is required that the reduced form is represented using DFT gates only. This representation is verified in HOL as follows:

**Theorem 6.**  $\vdash \forall A_1 B_1 C_1 D_1 E_1 W_1 G_1 H_1 I_1 J_1 K_1 L_1 N_1 O_1 P_1 A_2 B_2 C_2 D_2 E_2$   
 $W_2 G_2 H_2 I_2 J_2 K_2 L_2 N_2 O_2 P_2. \text{ALL\_DISTINCT} [A_1; B_1; C_1; D_1; E_1; W_1; G_1; H_1;$   
 $I_1; J_1; K_1; L_1; N_1; O_1; P_1; A_2; B_2; C_2; D_2; E_2; W_2; G_2; H_2; I_2; J_2; K_2; L_2; N_2; O_2; P_2] \Rightarrow$   
 $(Q_1 = \text{PAND } A_1 (I_1 + J_1 + K_1 + L_1) \cdot \text{PAND } (B_1 \cdot C_1 \cdot D_1 \cdot E_1 \cdot W_1 \cdot G_1 \cdot H_1) (I_1 + J_1 + K_1 + L_1) +$   
 $\text{PAND } A_2 (I_2 + J_2 + K_2 + L_2) \cdot \text{PAND } (B_2 \cdot C_2 \cdot D_2 \cdot E_2 \cdot W_2 \cdot G_2 \cdot H_2) (I_2 + J_2 + K_2 + L_2))$

## 5.2 Quantitative Analysis Results Using STORM

The quantitative analysis for the five benchmarks was conducted using STORM on a Linux machine with i7 2.4 GHZ quad core CPU and 4 GB of RAM. The efficiency of the proposed methodology is highlighted by analyzing the original DFTs and the reduced ones. In addition, the probability of failure for each DFT is evaluated for different time bounds, e.g. the probability of failure after 100 working time units. A summary of the analysis results are given in Table 2. It can be noticed that the number of states is reduced as well as the total analysis time. For the first benchmark, the analysis time is reduced due to the huge

reduction in the number of states. As mentioned earlier, many basic events are eliminated using the algebraic reduction theorems, which in turn reduced the total analysis time as well as the number of states. The reduction in the analysis time is also evident in the rest of the benchmarks, as given in Table 2. This is mainly because of two reasons, firstly, the number of states is reduced, and secondly, the original DFT is modeled as a Markov Automata (MA) as there is a non-deterministic behavior, while the reduced DFT is modeled as a Continuous Time Markov Chain (CTMC). This implies that in the reduced DFT the non-deterministic behavior caused by the failure dependency does not exist any more, as the reduction process depends on the time of failure of the gates, which allows solving the previously unresolved problems. We used STORM command (*firstdep*) to resolve the non-deterministic behavior in the original DFT to generate a CTMC instead of a MA. The results in Table 3 show that the number of states for the reduced DFTs is generally less than that of the original DFT with resolved dependencies, except for the HECS DFT, which requires further investigation. This emphasizes on the importance of the proposed methodology not only in providing a formal qualitative analysis but also in reducing the quantitative analysis cost in terms of time and memory, i.e., number of states. We believe that the proposed methodology can be implemented with any theorem prover that supports extended real data-type and with any probabilistic model checker that supports DFT analysis. In addition, we believe that applying this methodology to any DFT will reduce the analysis time as well as the number of states specially if the DFT has a non-deterministic behavior. This methodology can be enhanced if the model checker supports the temporal operators in addition to the supported FT gates. This means that we can use the result of the reduction directly without rewriting it in terms of FT gates. Moreover, the transformation process of the verified reduced DFT expression from the theorem prover script to its corresponding Galileo format can be automated to facilitate the overall analysis.

**Table 2.** STORM analysis results (before and after reduction)

DFT	Time		Before Reduction			After Reduction		
	Bound	#States	Analysis Time(sec)	Probability ofFailure	#States	Analysis Time(sec)	Probability ofFailure	
CPAND	1000	148226 (CTMC)	7.488	1.464103531e-4	66050 (CTMC)	3.032	1.464103348e-4	
ARHS	10	74 (MA)	169.81	0.00995049462	10 (CTMC)	0.067	0.009950461197	
	100	74 (MA)	*	**	10 (CTMC)	0.067	0.0954423939	
MCS	10	89 (MA)	139.7	0.01196434683	29 (CTMC)	0.061	0.01196434516	
	100	89 (MA)	*	**	29 (CTMC)	0.060	0.1166464887	
HECS	10	1051 (MA)	16359.83	0.01710278909	505 (CMTC)	0.123	0.01710276373	
	100	1051 (MA)	*	**	505 (CMTC)	0.123	0.1762782397	
HCAS	10	181 (MA)	275.31	2.000104327e-5	37 (CTMC)	0.070	2.99929683e-5	
	100	181 (MA)	*	**	37 (CTMC)	0.071	0.000300083976	

\*The analysis did not finish within 4 h.

\*\* No probabilities are recorded (analysis did not finish).

**Table 3.** STORM analysis results with resolved dependencies

DFT	Time	Dependency resolved in STORM			Algebraic Reduction		
	Bound	#States	Analysis Time(sec)	Probability of Failure	#States	Analysis Time(sec)	Probability of Failure
ARHS	10	10 (CTMC)	0.068	0.009960461197	10 (CTMC)	0.067	0.009950461197
	100	10 (CTMC)	0.1	0.09544239393	10 (CTMC)	0.067	0.0954423939
MCS	10	45 (CMTC)	0.064	0.01196434516	29 (CTMC)	0.061	0.01196434516
	100	45 (CMTC)	0.064	0.1166464887	29 (CTMC)	0.060	0.1166464887
HECS	10	379 (CTMC)	0.118	0.01710276373	505 (CMTC)	0.123	0.01710276373
	100	379 (CTMC)	0.121	0.1762782397	505 (CMTC)	0.123	0.1762782397
HCAS	10	73 (CTMC)	0.076	1.999530855e-5	37 (CTMC)	0.070	2.99929683e-5
	100	73 (CTMC)	0.076	0.0002001091927	37 (CTMC)	0.071	0.000300083976
	100000	73 (CTMC)	0.077	0.2772192934*	37 (CTMC)	0.074	0.3460009685*

\*The reported probability for the reduced DFT is closer to the probability reported in [3] for the same input failure distribution.

## 6 Conclusion

In this paper, we proposed a formal dynamic fault tree analysis methodology integrating theorem proving and model checking approaches. We first formalized the dynamic fault tree gates and operators in HOL theorem proving based on the time of failure of each gate. Using this formalization and the **extreal** library in HOL4, we also proved over 80 simplification theorems that can be used to verify the reduction of any DFT. We used these theorems to verify the equivalence of the raw and reduced DFTs using theorem proving. In addition, we provided a formally verified qualitative analysis of the structure function in the form of reduced cut sets and sequences, which, to the best of our knowledge, is another novel contribution. The quantitative analysis of the reduced structure function is performed using the STORM model checker. This ensures that the model checking results correspond to the original DFT, since we use the formally verified reduced DFT model for the quantitative analysis. Both the qualitative and quantitative analyses were conducted on five benchmark DFTs, providing formally verified reduced cut sets and sequences, as well as the corresponding probabilities of failure. In addition, the model checking results indicate that using the reduced DFT in the analysis has a positive impact on its cost in terms of both time and number of states. As a future work, we plan to provide the quantitative analysis of DFTs within HOL, which will allow us to have a complete framework for formal DFT analyses using theorem proving.

## References

1. Ruijters, E., Stoelinga, M.: Fault tree analysis: a survey of the state-of-the-art in modeling, analysis and tools. *Comput. Sci. Rev.* **15–16**, 29–62 (2015)
2. Stamatelatos, M., Vesely, W., Dugan, J., Fragola, J., Minarick, J., Railsback, J.: *Fault tree handbook with aerospace applications*. NASA Office of Safety and Mission Assurance (2002)

3. Merle, G.: Algebraic Modelling of Dynamic Fault Trees, Contribution to Qualitative and Quantitative Analysis. Ph.D. thesis, ENS, France (2010)
4. Dehnert, C., Junges, S., Katoen, J.-P., Volk, M.: A Storm is coming: a modern probabilistic model checker. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 592–600. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-63390-9\\_31](https://doi.org/10.1007/978-3-319-63390-9_31)
5. Volk, M., Junges, S., Katoen, J.P.: Fast dynamic fault tree analysis by model checking techniques. *IEEE Trans. Ind. Inf.* **14**, 370–379 (2017). <https://doi.org/10.1109/TII.2017.2710316>
6. Ahmad, W., Hasan, O.: Towards formal fault tree analysis using theorem proving. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) CICM 2015. LNCS (LNAI), vol. 9150, pp. 39–54. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-20615-8\\_3](https://doi.org/10.1007/978-3-319-20615-8_3)
7. Boudali, H., Crouzen, P., Stoelinga, M.: A compositional semantics for dynamic fault trees in terms of interactive Markov chains. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 441–456. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-75596-8\\_31](https://doi.org/10.1007/978-3-540-75596-8_31)
8. Merle, G., Roussel, J.M., Lesage, J.J., Bobbio, A.: Probabilistic algebraic analysis of fault trees with priority dynamic gates and repeated events. *IEEE Trans. Reliab.* **59**(1), 250–261 (2010)
9. Malhotra, M., Trivedi, K.S.: Dependability modeling using petri-nets. *IEEE Trans. Reliab.* **44**(3), 428–440 (1995)
10. Boudali, H., Dugan, J.: A new Bayesian network approach to solve dynamic fault trees. In: IEEE Reliability and Maintainability Symposium, pp. 451–456 (2005)
11. HOL4 (2017). [hol.sourceforge.net](http://hol.sourceforge.net)
12. Pullum, L., Dugan, J.: Fault tree models for the analysis of complex computer-based systems. In: IEEE Reliability and Maintainability Symposium, pp. 200–207 (1996)
13. Galileo. [www.cse.msu.edu/~cse870/Materials/FaultTolerant/manual-galileo.htm](http://www.cse.msu.edu/~cse870/Materials/FaultTolerant/manual-galileo.htm)
14. Arnold, F., Belinfante, A., Van der Berg, F., Guck, D., Stoelinga, M.: DFTCALC: a tool for efficient fault tree analysis. In: Bitsch, F., Guiochet, J., Kaâniche, M. (eds.) SAFECOMP 2013. LNCS, vol. 8153, pp. 293–301. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-40793-2\\_27](https://doi.org/10.1007/978-3-642-40793-2_27)
15. Ghadhab, M., Junges, S., Katoen, J.-P., Kuntz, M., Volk, M.: Model-based safety analysis for vehicle guidance systems. In: Tonetta, S., Schoitsch, E., Bitsch, F. (eds.) SAFECOMP 2017. LNCS, vol. 10488, pp. 3–19. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-66266-4\\_1](https://doi.org/10.1007/978-3-319-66266-4_1)
16. Ahmad, W., Hasan, O.: Formalization of fault trees in higher-order logic: a deep embedding approach. In: Fränzle, M., Kapur, D., Zhan, N. (eds.) SETTA 2016. LNCS, vol. 9984, pp. 264–279. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-47677-3\\_17](https://doi.org/10.1007/978-3-319-47677-3_17)
17. Elderhalli, Y.: DFT Formal Analysis: HOL4 Script and Storm Benchmarks (2017). [http://hvg.ece.concordia.ca/Publications/TECH\\_REP/DFT\\_TR17](http://hvg.ece.concordia.ca/Publications/TECH_REP/DFT_TR17)
18. Elderhalli, Y., Hasan, O., Ahmad, W., Tahar, S.: Dynamic Fault Trees Analysis using an Integration of Theorem Proving and Model Checking. Technical report, Concordia University, Canada (2017). <https://arxiv.org/abs/1712.02872>





# Twenty Percent and a Few Days – Optimising a Bitcoin Majority Attack

Ansgar Fehnker<sup>1</sup>(✉) and Kaylash Chaudhary<sup>2</sup>

<sup>1</sup> Formal Methods and Tools Group, University Twente,  
Enschede, The Netherlands  
ansgar.fehnker@utwente.nl

<sup>2</sup> School of Computing, Information, and Mathematical Sciences,  
University of the South Pacific, Suva, Fiji

**Abstract.** Bitcoin is a distributed online payment system that organises transactions into blocks. The size of blocks is limited to 1 megabyte, which also limits the number of transactions per second that can be confirmed. This year several attempts have been made to create a fork or a split that removes this restriction. One such alternative is Bitcoin Unlimited (BTU). Proponents of BTU have suggested to use a type of majority attack to force other Bitcoin miners to adopt BTU.

In this paper we model this attack in Uppaal, and analyse how long it will take for an attack to succeed, depending on the share the attacker has of the total network, and the so-called confirmation depth. The analysis shows that with a share of 20% an attack will be successful within a few days. This paper also looks at the effect of increasing the confirmation depth as a countermeasure.

## 1 Introduction

In circulation since 2009 [9], Bitcoin is the most popular digital currency. Bitcoin is managed by a peer-to-peer network. Every peer keeps a record of all transactions in a public ledger. Transactions are organised into separate blocks, all of which are linked to their immediate predecessor, forming a chain. The protocol uses a proof-of-work solution to induce a unique order on blocks, a process known as mining. As the difficulty of mining increased over the years, peers started working together in so-called pools.

Bitcoin has a block limit of 1 megabyte, which limits the number of confirmations to 3 transactions per second. This year has seen coin splits such as *Bitcoin Cash*, or forks, such as *SegWit*, aimed at this limitation. Proponents of one alternative, Bitcoin Unlimited (BTU), have suggested to use a type of majority attack to force adoption of BTU [11]. We will refer to this attack as the *Andresen* attack, after the former lead developer of Bitcoin who proposed the attack. Current lead developers of Bitcoin have argued that an attack on the main fork is a waste of computing resources [10]. That assessment will however depend on how many computing resources are required, and for how long.

This paper uses statistical model checking with Uppaal [7] to analyse the success chance, depending on the size of a mining pool. We use a parameter sweep to identify an optimised strategy, and then use this strategy for further analysis of potential counter measures.

Beukema [3] developed a formal model for double spending and TX corrupt peers. Andrychowicz et. al. modeled Bitcoin contracts with timed automata [2], and verified that an honest party can not lose Bitcoins. Herrmann considered the implementation, evaluation and detection of double-spending attacks [8]. This related work did not include blockchain forking, which is the focus of [6].

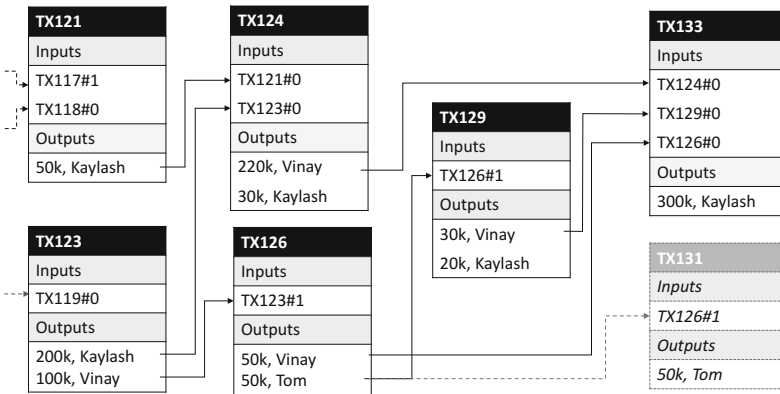


Fig. 1. Transaction graph

## 2 The Bitcoin Protocol

Bitcoin is a decentralised peer-to-peer electronic cash system, without central trusted authority [9]. Public/private key cryptography ensures the validity of transitions, while a so-called *mining* process determines the order of transactions.

*Transactions.* There are two types of transactions: coin-base and regular transactions. In this paper we consider regular transactions, transferring existing Bitcoins from one user to another.

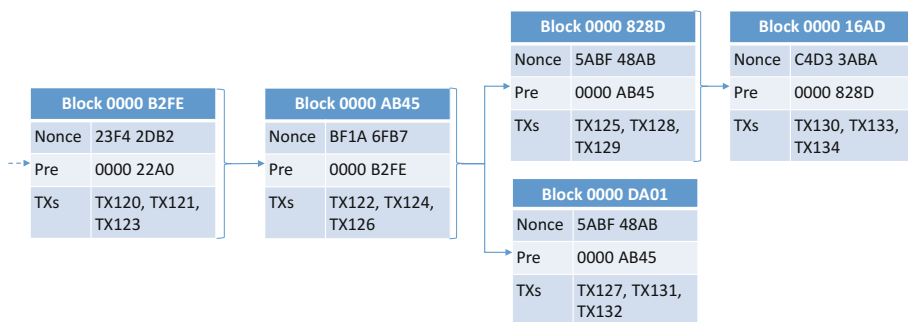
Each transaction has one or more transaction inputs and one or more outputs. An input is a reference to an output of a previous transaction. It proves that the senders possess the Bitcoins they claim to have. The transaction output specifies an amount and a recipient.

Figure 1 gives an example of a transaction graph. Transaction TX124 has two inputs; transactions to user Kaylash worth 250k, and two outputs: 220k to user Vinay, and 30k to user Kaylash<sup>1</sup>. The first output is in turn an input of transaction TX133, and thus spent. The second has not been used and is thus unspent.

<sup>1</sup> The amount is expressed in Satoshi (1 BTC is 100 000 000 Satoshi).

To guard against double spending each output can only be used once. TX129 and TX131 cannot both be part of the transition graph, since both spend the second output of TX126. To impose an order on transactions – to decide which transaction came first – the Bitcoin uses a so-called *blockchain*.

*Blockchain.* A block contains a set of transactions, a header, and the hash of the predecessor block. Transactions in the same block are considered to have happened at the same time. Transactions in different blocks are ordered using the predecessor relation between blocks. Transactions are only confirmed if they appear in some block; unconfirmed transactions are kept in the *transaction pool*. A peer selects transactions from the transaction pool for a new block at the end of the block chain, provided it completes a so-called *proof-of-work*.



**Fig. 2.** Blockchain

For the *proof-of-work* the node randomly selects a *nonce*, which will become part of the block header. The hash of the block (including this nonce) is calculated and the result is compared with the *target value*, set by the Bitcoin network. If the hash is lower than the target value, the proof-of-work is completed; otherwise the node repeats this process. Once a nonce is found, the block becomes valid and is broadcasted to the network. This process is called *mining*.

With different *miners* working on different blocks, the blockchain may have forks, i.e. that two blocks are created and broadcasted over the network simultaneously. Some peers might receive the first block first, and others the second. Peers continue with the block they received first.

Transactions in the longest chain are considered confirmed. Other transactions are added back to the transaction pool, and can be used to build new blocks. Miners will usually extend the longest chain, because they will only be rewarded for blocks in the longest chain. There is a non-negligible probability that a fork of the longest chain will become the longest. The distance of a block to the end of the chain is called the confirmation depth. It is advised to only consider transactions in blocks with confirmation depth 6 or more settled [4].

Figure 2 depicts a blockchain that includes the transaction of Fig. 1. Transaction TX126 is included in Block 0000 AB45, and TX129, which uses an output

of 50k for Tom from *TX126*, is in *Block 0000 828D*. *TX131* cannot be included in this block, or any of its successors, since any output can only be used once. *TX131* could however be included in a fork of *Block 0000 AB45*. If this fork becomes the longest, *TX131* would be considered valid, instead of *TX129*. The 50k would have gone to Tom, instead of Vinay and Kaylash. However, for this to happen the miners would have to outcompete the rest of the network which will extend the longest chain ending in *Block 0000 16AD*.

*Hash-Rate.* The network *hash-rate* (hashes per second) is a measure for the processing power of the Bitcoin network. The *target value* is adapted every 2016 blocks, to achieve a desired confirmation time. In 2017 the confirmation time was about 12 min [1].

The hash-rate of a pool is relative to hash-rate of the network. A hash-rate of  $r \in [0, 1]$ , means that the pool alone would find 1 block in  $12/r$  min. At the time of writing, November 2017, the largest pool *AntPool* had a hash-rate of 18%. These numbers are subject to fluctuation; in 2014 pool *Ghash.io* came close to a hash rate of 50% [5].

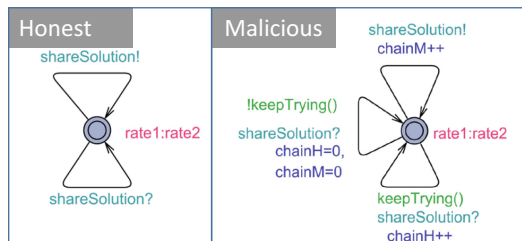
*Andresen Attack.* Andresen proposed that BTU miners could create a fork, mine it in secret until it reaches a length of 11, and then publish the fork at once [11]. Previously confirmed transactions in the honest fork would become unconfirmed, and Bitcoin miners would lose their reward for extending the previously longest fork. The aim is to undermine the trust in classic Bitcoin. The next sections investigate at what hash-rate of the malicious pool this attack becomes feasible.

### 3 Model and Strategies

The attack is modelled as a race between the honest pool and the malicious pool. This model is simplified by the fact the malicious pool will work exclusively on its fork, while the honest pools work exclusively on their fork, the publicly known blockchain. The model does not have to take into account network delays or concurrent mining on the same fork.

The model, to the right, contains two integer variables, `chainH` and `chainM`, to measure the length of the honest and malicious fork. Both pools announce solutions on channel `shareSolution`, with `rate1:rate2`.

If the malicious pools detects that the honest pool found a block it either continues the current attack, i.e. with the current fork, or abandons this attempt, and starts a new fork. Continuing is modelled as incrementing `chainH`. Starting a new fork is modelled by resetting `chainH` and `chainM` to zero.

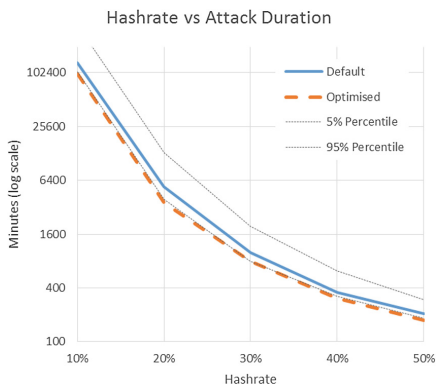


The strategy deciding to continue or abandon an attack is defined by an array `int threshold[7]`. The race will continue if `chainH < threshold[chainM]`, and will be abandoned otherwise. The attack is successful if the malicious fork reaches the confirmation depth 6, **and** if at the same time the length of the malicious fork exceeds the length of the honest fork. This is expressed as the following Uppaal property:

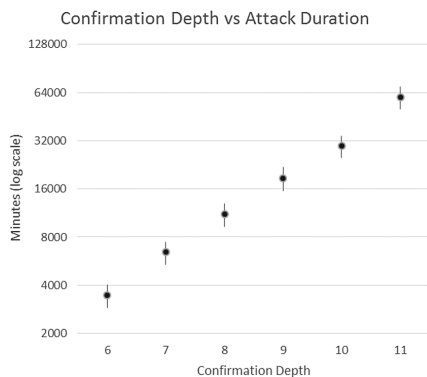
$$\text{Pr}[\leq 1000000] (\langle \rangle \text{chainM} \geq 6 \text{ and } \text{chainM} > \text{chainH}) \quad (1)$$

## 4 Analysis

The analysis systematically sweeps through parameters for `int threshold[7]`. It considers values from 0 to 10, with a difference of at most 4 between `chainM` and `threshold[chainM]`. This leaves 12597 arrays to consider for an optimised strategy. The analysis computed the expected attack duration for hash-rates from 10% to 50% for each of these 12597 candidates. It identified an *optimised* strategy, defined by threshold array `[0, 2, 4, 5, 7, 8, 9]`, which is the rounded average of the top 1% of parameter values. This strategy means that the malicious pool should continue with the current attempt, even if it trails in the race, as long as it found a few blocks itself.



**Fig. 3.** Duration of an attack for different hash-rates for the default and optimised strategy.



**Fig. 4.** Duration of attack for different confirmation depths. Results for 20% hash-rate.

Figure 3 compares the duration of the optimised strategy with the default strategy that abandons an attempt if the honest fork exceeds the malicious fork (array `[0, 1, 2, 3, 4, 5, 6]`). At a hash-rate of 10% the optimised strategy has an expected duration of 100203 min (69d:14h:03m), at hash-rate 20% this is 3582 min (2d:11h:42m). The default strategy is expected to take 134023 min (93d:1h:43m) and 5403 min (3d:18h:03m) respectively. At a hash-rate of 20% (or more) the Andresen attack appears to be feasible.

One counter measure would be to increase the confirmation depth. Figure 4 shows for a hash-rate of 20% that increasing the confirmation size by one increases the estimated duration on average by 77%. Note that the scale is logarithmic. For confirmation depth 10 this means that the expected duration is 29600 minutes (20d:10h:20m), and for depth 11 – as mentioned initially by Andresen – it increases to 60024 minutes (41d:16h:24m). Increasing confirmation depth increases the time to confirm transactions. A confirmation depth of 10 instead of 6, for example, means that it will cost 80% more time.

All results have been computed with Uppaal 4.1.19, at confidence level 0.99.

## 5 Conclusion

Analysis with Uppaal SMC of the Andersen attack shows that it does not require the majority hash-rate to succeed. A hash-rate of 20% would yield a success within a few days, sufficient to construct a malicious fork of length 6, something that would be unprecedented. Since the malicious pool tries to catch up from behind if it trails, the malicious fork will even have length 7 or more in 11% of the cases. Classic Bitcoin miners could adopt a larger confirmation depth, but that would affect the entire network and the efficiency of the currency as a whole. To mount a counterattack classic miners would actually have to adopt BTU, which is the declared aim of the initial attack.

The analysis does not depend on particularities of BTU. Similar considerations can be made for other coin forks. Bitcoin protocol developer Mark Corallo argued in [10] that an attack on the main fork is unlikely, given that resources could be better used for mining the main fork. Considering that some of the players have or are near the required hash-rate, and that optimising the strategy reduces the cost of an attack significantly, may change this economic argument. The cost of an attack may weigh up against long term benefits of setting a new standard, or short term benefits of intentionally upsetting the market.

Uppaal SMC's input language made it easy to model the attack, and its specification language was sufficiently expressive to analyse the model. Using these we were able to do a parameter sweep to optimise a strategy for the Andresen attack. The optimised strategy significantly reduces the expected duration of an attack, and thus resources required for it. The model, with additional analysis results are made available on <http://wwwhome.ewi.utwente.nl/~fehner/V17/>.

## References

1. Bitcoin charts (2017). <https://blockchain.info/charts>
2. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, Ł.: Modeling bitcoin contracts by timed automata. In: Legay, A., Bozga, M. (eds.) FORMATS 2014. LNCS, vol. 8711, pp. 7–22. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-10512-3\\_2](https://doi.org/10.1007/978-3-319-10512-3_2)
3. Beukema, W.: Formalising the bitcoin protocol. In: 21th Twente Student Conference on IT (2014)

4. Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J.A., Felten, E.W.: Research perspectives and challenges for bitcoin and cryptocurrencies. IACR Cryptology ePrint Archive 2015, 261 (2015)
5. Cawrey, D.: Are 51% Attacks a Real Threat to Bitcoin? CoinDesk, June 2014. <https://www.coindesk.com/51-attacks-real-threat-Bitcoin/>
6. Chaudhary, K., Fehnker, A., van de Pol, J., Stoelinga, M.: Modeling and verification of the bitcoin protocol. In: MARS 2015. EPTCS (2015)
7. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B.: STTT. Uppaal SMC tutorial **17**(4), 397–415 (2015)
8. Herrmann, M.: Implementation, evaluation and detection of a double-spend-attack on Bitcoin. Master's thesis, Master Thesis ETH Zürich, April 2012
9. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2009). <http://www.Bitcoin.org>
10. Shin, L.: What Will Happen At The Time Of The Bitcoin Hard Fork? Forbes, October 2017. <https://www.forbes.com/sites/laurashin/2017/10/31/what-will-happen-at-the-time-of-the-bitcoin-hard-fork/>
11. Van Wirdum, A.: Bitcoin Unlimited Miners May Be Preparing a 51% Attack on Bitcoin. Bitcoin Magazine, March 2017. <https://Bitcoinmagazine.com/articles/Bitcoin-unlimited-miners-may-be-preparing-51-attack-Bitcoin/>



# An Even Better Approach – Improving the B.A.T.M.A.N. Protocol Through Formal Modelling and Analysis

Ansgar Fehnker<sup>1</sup> , Kaylash Chaudhary<sup>2</sup>, and Vinay Mehta<sup>2</sup>

<sup>1</sup> Formal Methods and Tools Group, University Twente, Enschede, The Netherlands  
[ansgar.fehnker@utwente.nl](mailto:ansgar.fehnker@utwente.nl)

<sup>2</sup> School of Computing, Information, and Mathematical Sciences,  
University of the South Pacific, Suva, Fiji

**Abstract.** This paper considers a network routing protocol known as *Better Approach to Mobile Adhoc Networks* (B.A.T.M.A.N.). The protocol has two aims: first, discovery of all bidirectional links, and second, identification of the best-next-hop to the other nodes. A key mechanism of the protocol is to flood the network at regular intervals with so-called *originator messages*.

In previous work we formalised the B.A.T.M.A.N. protocol in Uppaal and found several ambiguities and inconsistencies [2]. More importantly, explicit choices in the RFC had, unfortunately, a negative impact on route discovery. This previous work compared a literal model based of the RFC with an incremental improvement. This paper goes one step further and proposes an alternative that departs from the RFC. We compare the performance using simulations in Uppaal, for static as well as dynamic topologies. The analysis shows that the proposed alternative reduces the number of suboptimal routes significantly, and recovers better from routing errors that are introduced by mobility.

## 1 Introduction

The German “Freifunk” community developed the network routing protocol *Better Approach to Mobile Adhoc Networks* (B.A.T.M.A.N.) as an alternative to OLSR. B.A.T.M.A.N. is a proactive protocol for detecting bidirectional links and identifying the best-next-hop to all other nodes. Each node floods the network at regular intervals with so-called originator messages and records information on received originator messages in a so-called *sliding window*.

Previous work [2] revealed that the RFC [13] contained several ambiguities and inconsistencies, as well as a few explicit choices that had a negative impact on the route discovery process. The analysis in [2] used model checking and simulation of a static network to investigate occurrence of suboptimal routes.

This paper goes one step further and proposes a model that departs from the RFC. It proposes a new sliding window that stores the time-to-live values



of received messages. This information is then used to identify the best-next-hop. The alternative model (1) redefines the “best-next-hop” to also include alternative routes, (2) updates the sliding window if it finds a better route, (3) forwards only messages that are evidence of *improved routes*, and (4) compares the time-to-live only of messages that have the same sequence number. The model in [2] only addressed the first point fully, the next two partially, and the last one not at all. The analysis in [2] considered only static topologies, while this paper also analyses the performance in the presence of a mobile node.

The performance of B.A.T.M.A.N. has been analysed in a real environment in [11,14]. A simulation of a real environment has been used to analyse B.A.T.M.A.N. in [8,12]. Furlan simulated the performance of B.A.T.M.A.N. for a select type of network topologies [6]. These studies have in common that they consider a particular implementation with specific hardware in a specific environment and do not analyse the routing algorithm in isolation. This paper instead uses a formal timed automaton model of the protocol and its proposed alternative. This allows us to study the protocol in isolation, including non-deterministic timing, interleaving of message processing in concurrent nodes, and a probabilistic model of mobility.

A formal analysis of B.A.T.M.A.N. was conducted by Cigno and Furlan in [7]. This analysis discovered that routing loops are possible, and it proposed improvements that ensure loop freedom. This work was complemented by simulation studies and real measurements. While this work does not use model checking per se, it presents a formal model to study routing loops. In comparison, this paper presents a formal Uppaal model, and uses verification and simulation to study the quality of route discovery. Uppaal has been used previously to study other protocols, such as LUNAR, OLSR, DYMO, AODV, and LMAC [1,4,5,10,15].

Section 2 describes the B.A.T.M.A.N. routing protocol and the two different models. Section 3 describes the Uppaal model, and Sect. 4 the results of the performance analysis. Section 5 concludes the paper.

## 2 The B.A.T.M.A.N. Protocol

The German “Freifunk” community has developed the network routing protocol known as Better Approach to Mobile Adhoc Network [13]<sup>1</sup> as an alternative to OLSR (Optimized Link State Routing Protocol) [3]. Instead of having each node compute routing tables that capture the complete network topology as OLSR does, B.A.T.M.A.N. nodes maintain only information on neighboring nodes. This information is used to identify the best-next-hop [9].

Each node broadcasts at regular intervals *originator messages* (OGMs). These are forwarded to neighbouring nodes, if they meet certain conditions, which will be discussed later. Each node keeps a record of received OGMs, first to correctly identify all available bidirectional links, and then to find the best-next-hop to a destination. The protocol is phrased in terms of the origin of the

<sup>1</sup> The RFC abbreviates the protocol as B.A.T.M.A.N., including the dots.

OGMs; once routes have established these originators will be the destination for packets. It is a desired property that the so-called best-next-hops will indeed realize the best route, and that the protocol recovers quickly from topology changes.

Formalisation of the RFC in [2] as an Uppaal model revealed a number of ambiguities, for example the distinction between a new OGM and an OGM that is not a duplicate. To resolve these ambiguities, that paper used a model for a small network and verified a few basic properties, such as successful route discovery. None of the choices that had to be made to resolve those ambiguities was surprising, and any implementation of B.A.T.M.A.N. will likely resolve these in a similar way.

However, the formalisation also revealed that RFC also contained few explicit choices that arguably undermine the route discovery process. This means the RFC explicitly defines a rule or process that is arguably counterproductive. To investigate the effect of those choices this paper will present a *Literal model* that adheres to the RFC, and an *Alternative Model* that strives to improve on the route discovery process. We will present both models alongside each other.

The RFC defines the message format of OGMs, a sliding window to record OGMs, rules for processing of OGMs, and the processes to update the sliding window, ranking the neighbors, and for assembling OGMs that need to be forwarded. Table 1 summarizes these rules and processes for both the *Literal model* and *Alternative Model*.

## 2.1 Literal Model

An OGM contains an *originator ID* (OID), a *sender ID* (SID), a *sequence number*, a *time-to-live* (TTL), and two flags: the *uni-directional link flag*, and the *direct-link-flag*. The TTL is used as a measure for the distance travelled. The last two flags are used for the *bidirectional-link check*. We omit further discussion of the bidirectional-link check and assume that it works as desired. It is mentioned here, since the RFC intertwines it with route discovery. For more details see [2].

Each node maintains a *sliding window* to record OGMs. It specifies for each originator a range of recent sequence numbers. The window is used to record for each sender if it has sent a qualifying OGM with an in-window sequence number. If an OGM with a newer sequence number is processed, the range will move accordingly. In addition, each node keeps for each originator the last TTL as a measure of the distance to the originator, which will be updated each time an OGM is recorded.

The RFC defines the *best-next-hop* as the sender with the most recorded OGMs within the sliding window. It specifies that at any time there is only one designated best-neighbor, even if multiple nodes have the same ranking. This designation will only change if another node surpasses the ranking of currently designated best-next-hop.

The condition for forwarding an OGM in the literal model (Table 1) is meant to ensure that only OGMs that arrive via a best route are rebroadcasted, and that they neither are a duplicate, nor have been forwarded before. If a neighbor

**Table 1.** Comparison of literal and alternative B.A.T.M.A.N. models.

		Literal Model	Alternative Model
<i>Generate OGM</i>		Generate OGMs at regular intervals. Increment the sequence number modulo upper bound, set sender ID and OID to own ID, TTL to maximum, and the other flags to <i>false</i> .	
Sliding Window	<i>Record</i>	A mapping for each sender from in-window sequence numbers to a Boolean.	A mapping for each sender from in-window sequence numbers to TTL.
	<i>Ranking</i>	Rank by most in-window sequence numbers.	Rank by most best TTL per in-window sequence number.
	<i>Best-next-hop</i>	Designate one best-next hop. Change only when required.	Treat all currently top ranked nodes as best-next-hops.
Update	<i>Condition</i>	Link to sender is bidirectional, and OGM has a newer sequence number.	Link to sender is bidirectional, and OGM has a newer sequence number, or, the sequence number is in-window and the TTL is better than or equal to the best TTL for that sequence number.
	<i>Process</i>	Shift the window if the sequence number is newer. Record the in-window sequence number, and update the last TTL.	Shift the window if the sequence number is newer. Record the TTL for in-window sequence number.
Forwarding	<i>Condition</i>	Link to sender is bidirectional, and sender is the best-next-hop, and the TTL is larger than 2, and the OGM is not a duplicate or has same TTL as the last recorded TTL.	Link to sender is bidirectional, and sender is the best-next-hop, and the TTL is larger than 2, and sequence number is newer, or it is in-window and the TTL is strictly better than the best TTL for that sequence number.
	<i>Process</i>	Set sender ID to own ID, decrement TTL, set <i>isDirect</i> to true if previous sender is equal to originator, set <i>isUniDirectional</i> to true if the link to sender is unidirectional.	

OID	Last TTL	SLIDING WINDOW					
B	10	SID <sub>↓</sub>	Sequence Numbers				
			13	14	15	0	1
		B		1		1	1
		C	1				
		D			1		
C	8	SID <sub>↓</sub>	Sequence Numbers				
			6	7	8	9	10
		B				1	1
		C		1			
		D	1		1		
D	7	SID <sub>↓</sub>	Sequence Numbers				
			2	3	4	5	6
		B		1		1	1
		C					
		D	1		1		

OID	Last TTL	SLIDING WINDOW					
B	10	SID <sub>↓</sub>	Sequence Numbers				
			13	14	15	0	1
		B		1		1	1
		C	1				
		D			1		
C	7	SID <sub>↓</sub>	Sequence Numbers				
			7	8	9	10	11
		B			1	1	
		C	1				
		D		1			1
D	7	SID <sub>↓</sub>	Sequence Numbers				
			2	3	4	5	6
		B				1	1
		C		1			
		D	1		1		

Fig. 1. Sliding window for Node A in the literal model, before and after an update.

rebroadcasts an OGMs that the node has sent previously, it will have a different (lower) TTL than the original.

Note that the rules for updating the sliding window and for forwarding OGMs are not mutually exclusive; some OGMs may be only used for the update, some may only be forwarded, and others for both. If none of these rules apply, and none of the bidirectional link rules, the OGM will be silently dropped.

*Example.* Figure 1 gives an example of a sliding window for a node A in a network with four nodes. For each originator, nodes B to D, it records which node forwarded an OGM from that originator. For example, the sliding window (before the update) recorded that node C forwarded an OGM from node B with sequence number 13 to node A. For originator B we have that 3 out of 5 OGMs in the sliding window were submitted by B – more than from C and D, hence B is the designated best-next-hop. For originator C, nodes B and D have both sent 2 in-window OGMs. The protocols designates one of them as best-next-hop and only changes it if another node actually surpassed the ranking of the currently designated best-next-hop.

If an OGM with a newer sequence number is processed, the range will move accordingly. For example, if node A receives from node D an OGM with OID C, TTL 7, and sequence number 11, it will move the window from sequence numbers 6 to 10, to sequence numbers 7 to 11. All entries will be shifted, and D will be recorded as having sent an OGM, and the last TTL will be set to 7 (Fig. 1 after). If it receives another OGM with OID C and sequence number 11, but from node B, it will not be recorded since it is a duplicate.

The decision whether to forward an OGM also takes into account whether the best-next-hop sent it. An OGM will only be forwarded if it was sent by the designated best-next-hop. For example, assume that node B is the current best-next-hop for OID C, for Fig. 1 before the update. If node A receives from node D an OGM with OID C, TTL 7, and sequence number 11, it will update the window as explained in the previous paragraph. However, it will not forward it, since it was sent by node D, which is not the designated best-next-hop. The

intention is to not forward OGMs that were received via sub-optimal routes, in this case however it drops an OGM received from a node that has the same ranking as the designated best-next-hop.

The decision whether to forward an OGM also takes into account the TTL. An OGM with a TTL of value 2 will be dropped.<sup>2</sup> The purpose of this rule is to limit the number of hops an OGM travels. The TTL is however also used to compare an OGM, with the last TTL. Suppose, we continue the above scenario, and node **B** sends an OGM to node **A** with OID **C**, sequence number 11 and TTL 8. It was sent from the best-next-hop, however, the OGM is a duplicate and the TTL is different from the last TTL. Hence it will not be forwarded. This even though the TTL is higher than the last TTL, i.e. the OGM actually travelled a shorter distance than the last recorded OGM.

## 2.2 Alternative Model

The alternative model retains the basic structure, but redefines the rules to improve the route discovery process. The essential difference is that this model records the TTL of each in-window OGM, not just of the last. Figure 2 gives an example of routing information maintained by the alternative model. Table 1 gives a summary of the important changes.

The alternative model ranks the senders by how many OGMs had a maximal TTL for any given in-window sequence number. All nodes that lead the ranking are considered a best-next-hop. Furthermore, the alternative model extends the condition for updating. It also records an OGM if it has a TTL that is better or equal to the currently best TTL for that sequence number. This means it also records OGMs that arrive via alternative routes that are as good **or better**.

Furthermore, the alternative model redefines the condition for forwarding. It will rebroadcast an OGM only if it has a better (thus different) TTL than previously received OGMs with that sequence number. Unlike the literal model it will not forward OGMs that have the same TTL. However, unlike that model, it **will** forward only those with a **strictly better TTL**.

Note also that the literal model compares any TTL with the last TTL, regardless of sequence number. This can be problematic if messages do not arrive in order, and the best TTL may have changed in the meanwhile. The alternative model only compares TTL from OGMs with the same sequence number. It is still possible that messages do not arrive in order, but the effect of topology changes are confined to one sequence number.

To summarize: The alternative model (1) redefines the “best-next-hop” to also include alternative routes, (2) updates the sliding window if it finds better routes, (3) forwards only messages that are evidence of *improved routes*, and (4) only compares the time-to-live of messages that have the same sequence number.

---

<sup>2</sup> The RFC states that the OGM should be dropped if the TTL after decrementing becomes 1.

Alternative Model - Before							
OID	SLIDING WINDOW						
B	SID <sub>↓</sub>	Sequence Numbers					
		13	14	15	0	1	
	B		10			10	10
	C	8					
	D	8		8			
C	SID <sub>↓</sub>	Sequence Numbers					
		6	7	8	9	10	
	B	8		8	8	8	
	C		10				
	D	8		8	9		
D	SID <sub>↓</sub>	Sequence Numbers					
		2	3	4	5	6	
	B		8		7	7	
	C				8		
	D	10		10			

Alternative Model - After							
OID	SLIDING WINDOW						
B	SID <sub>↓</sub>	Sequence Numbers					
		13	14	15	0	1	
	B		10			10	10
	C	8					
	D	8		9			
C	SID <sub>↓</sub>	Sequence Numbers					
		7	8	9	10	11	
	B		8	8	8	8	
	C	10					
	D		8	9		7	
D	SID <sub>↓</sub>	Sequence Numbers					
		2	3	4	5	6	
	B		8		7	7	
	C				8		
	D	10		10			

Fig. 2. Sliding window for Node A in the literal model, before and after an update.

Example. Figure 2 gives an example of a sliding window for node A. Before the update, both nodes B and D are considered best-next-hops for OID C, since they both have the highest TTL for 3 out of the last 5 sequence numbers.

Suppose node A receives an OGM from node D with OID C, TTL 7, and sequence number 11. The sliding-window will move, and sequence number 7 will be recorded. The OGM will also be forwarded, since it is a new OGM, and node D is a best-next-hop. Suppose further that node B forwards another OGM to node A, with OID C and sequence number 11, now with TTL 8. This OGM will also be recorded and forwarded, since it improves on the previous OGM with sequence number 11. Node B will now be the next-best-hop; it has three times the best TTL, node D only twice, and node C only once.

The next sections analyse the effect of these changes on the performance, in particular on the number of suboptimal routes and on the recovery from route errors due to mobility.

### 3 Uppaal Model

For the performance analysis we use a timed (and probabilistic) Uppaal model. It uses a single template for all nodes in the network, and the entire network is a composition of  $N$  nodes.

The node template has four control locations. The first is a committed initial location, the second, labelled **Empty**, models a node that has no OGM in its message buffer, the third, labelled **Processing**, models a node that has OGMs in its buffer, and, finally, another committed location. This location is entered after processing of an OGM, and is used to check if the buffer is empty or not. The edge from the initial location to location **empty** initialises the sliding windows and other local variables.

From location **Processing** there are 7 outgoing edges. They model sending and receiving of OGMs, and the processing of the OGMs:

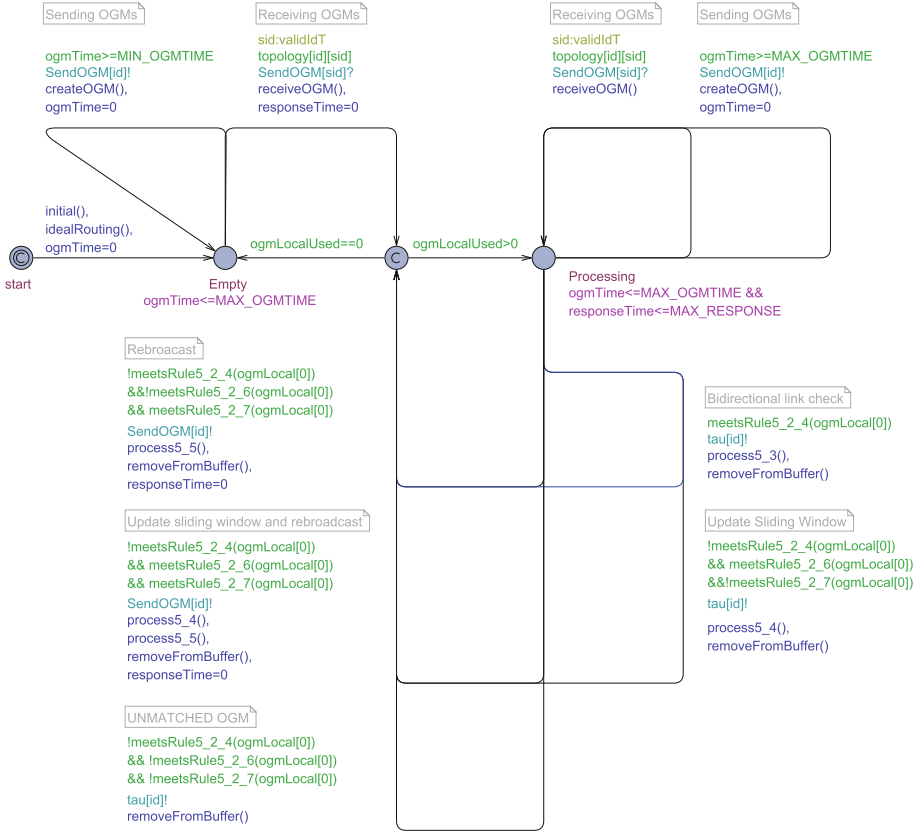


Fig. 3. Timed model for simulation

- An edge that synchronizes on channel `SendOGM[oid]!`. This transaction creates an OGM, and copies it to the shared global variable `ogmGlobal`.
- An edge that synchronizes on channel `SendOGM[sid]?`, where `sid` is a valid sender ID. The guard `topology[sid]` ensures that sender and receiver are connected. This transaction calls a function `receiveOGM()`, which copies the OGM from global variable `ogmGlobal`, and appends it to the local buffer `ogmLocal`.
- Five edges that model the processing of OGMs in the buffer. The different rules and processes refer to sections in the RFC.
  - *Rule 5.2.4* and *Process 5.3* relate to the bidirectional-link check.
  - *Rule 5.2.6* and *Process 5.4* relate to updating the sliding window.
  - *Rule 5.2.7* and *Process 5.5* relate to forwarding OGMs.

If Rule 5.2.7 applies, the edge will synchronize on channel `SendOGM[oid]!`, i.e. it forwards an OGM. All others are labelled with broadcast channel `tau[id]`. These edges model a local update of the routing information. The label allows

us to give those transition a higher priority – to reduce the number of states – while it will not affect the routes that will be discovered.

The model uses clock `ogmTime` to ensure that any node sends an OGM once between `MIN_OGMTIME` and `MAX_OGMTIME`. The transition that models creation and sending of new OGMs includes guard `ogmTime >= MIN_OGMTIME`, and all non-committed control locations invariant `ogmTime <= MAX_OGMTIME`.

The model uses clock `responseTime` to ensure that a node rebroadcasts within `MAX_RESPONSE` time units, while there is an OGM in the buffer. The model includes two control locations, `Empty` for when the buffer is empty, `Processing` for when the node processes OGMs. In the latter location there is the additional invariant `responseTime <= MAX_RESPONSE`, which enforces a timely response.

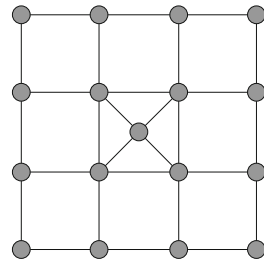
The model also includes a number of auxiliary functions to support analysis of the performance by simulation, such as function `countRouteMismatch()` which counts how many “best-next-hops” are actually suboptimal.

The model for a network with a dynamic topology includes one more template that modifies the topology at random, based on an underlying random walk model, as described in [5].

This basic structure is the same for the literal and the alternative model. The differences given in Table 1 is implemented by different implementations of the functions that check conditions and perform updates.

## 4 Simulation Results

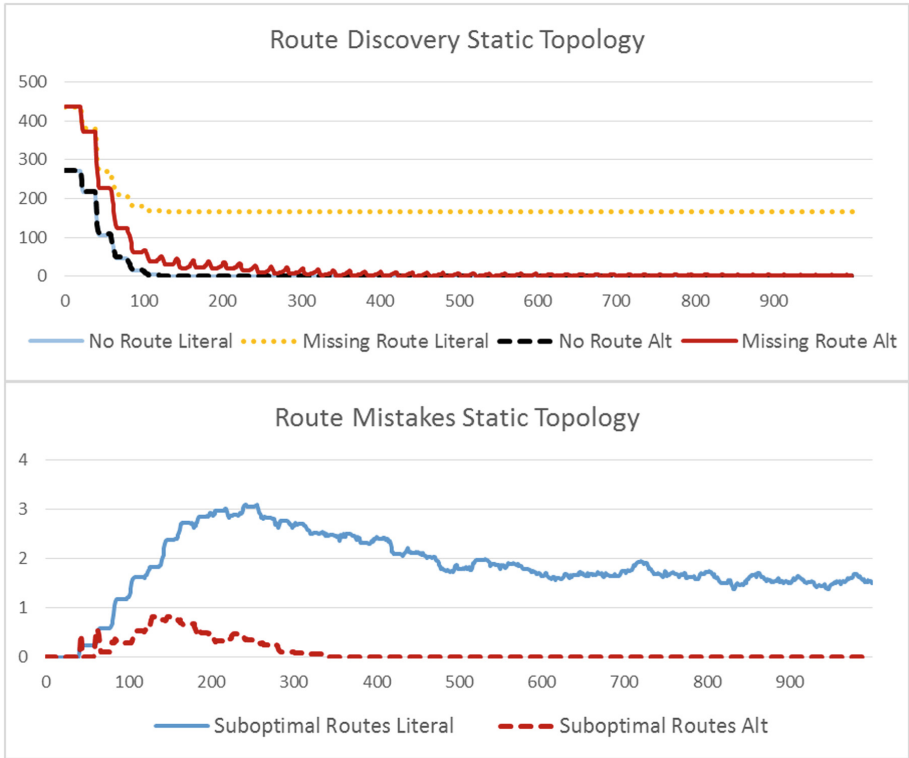
This section compares the two models using simulation with Uppaal. This comparison considers first a static topology to evaluate the speed of route discovery and incidences of suboptimal routes. It then considers a dynamic topology with one moving node for comparing the performance when links are deleted or added. The simulations use a 4 by 4 grid plus one additional central nodes, as depicted to the right. Unlike a linear topology or a fully connected graph, this setup has many potential sub-optimal routes.



Processing of OGMs takes at most 1 time unit, new OGMs are created once between 19 and 20 time units, and each simulation stops after 1000 time units. The size of the sliding window is 5. We use an Intel i5-5200 CPU 2.2 Ghz processor with 8 GB RAM running Uppaal 4.1.19. A single simulation of the model with dynamic topology takes about 1200 s. All results are for 100 runs.

**Route Discovery.** Figure 4 depicts the results route discovery for a static topology. It shows for how many node/originator pairs *no route*, i.e. no best-next hop has been identified, and also for how many pairs a potential best-next-hop is missing. The latter applies if more than one optimal route to the originator exists.



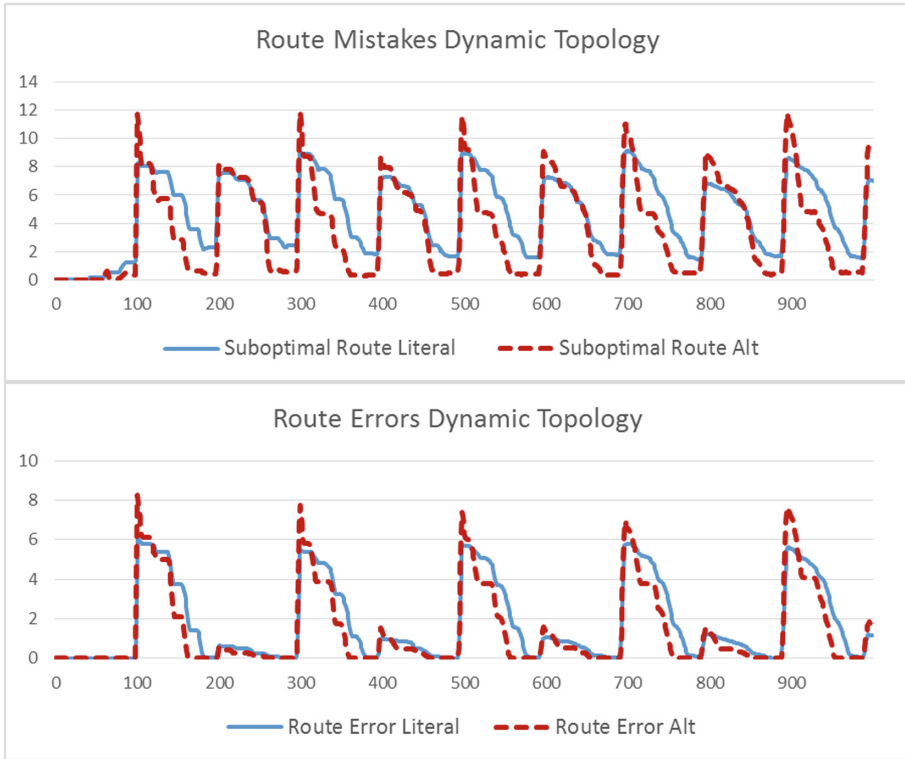


**Fig. 4.** Results on route discovery and route mistakes for static topology. Averages over 100 runs.

Both versions discover within 120 time units at least one route for each node/originator pair. It takes 6 rounds of OGMs, which corresponds to the network diameter 6. The literal model converges to 164 missing routes. It finds no additional routes, since, by design, only one best-next-hop is designated. In the alternative this number will decline to almost zero, except for the disturbance introduced by new OGMs every 20 time units.

More telling are the route mistakes, i.e. the number of suboptimal routes. The literal model number converges to an average of 1.5 suboptimal routes at time 1000, while the alternative model converges to none. Note that the literal model chooses only one best-next-hop; if that hop is suboptimal it has no optimal alternative.

**Error Recovery.** Figure 5 shows results for a dynamic topology. The central node is mobile, which means moves freely in the entire 4-by-4 grid, adding and deleting links with a probability that depends on the current topology. The underlying model for mobility is a random walk, as described in [5]. There will be one change to the topology every 100 time units. Figure 5 gives the number of suboptimal routes and the number of route errors. The latter are best-next-hops

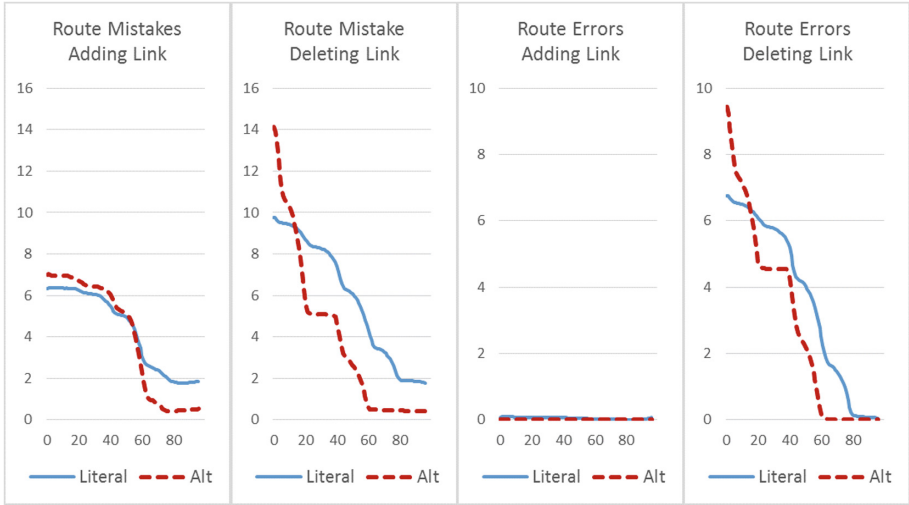


**Fig. 5.** Results on route mistakes and route error for dynamic topology. Averages over 100 runs.

that are actually not even directly connected to the node. Route errors happen when a link is deleted due to mobility. The results show that the alternative model will recover faster from suboptimal routes and route errors.

Figure 5 combines adding and deleting nodes in one graph. Over the 100 runs a link was added 411 times and deleted 489 times, but in Fig. 5 they are aggregated the same way. The difference between the route errors in the interval 100 to 200 compared to the interval 200 to 300 is explained by the fact that – due to the geometry of the topology – the mobile node will more often delete a link at time 100, and more often add a link at time 200.

To isolate the effects of link addition and deletion, we looked at whether at a beginning of a 100 times unit interval a link was added or deleted, and aggregated those results separately. Figure 6 separates the results for adding and deleting links, for a clearer picture. The results show that the alternative model has higher initial values for suboptimal routes and route errors. This is because of multiple best-next-hops in the alternative model. It identifies on average 60% more. If a link changes, it affects more best-next-hops. On the other hand it also offers more alternatives, which allows it to recover faster.



**Fig. 6.** Recovery after link deletion or addition. Values are averages.

Figure 6 shows that the alternative model recovers faster from route errors when a link is deleted, and converges faster to fewer suboptimal routes when a link is added. The small number of route errors for an added link is noise remaining from earlier deletions. Adding a link may introduce suboptimal routes, but not route errors.

**OGM Classification.** The previous results compare the performance of both models in terms of route mistakes and errors. Another measure is to see how many OGMs are created, forwarded and recorded. Figure 7 compares how received OGM were classified; whether to perform a bidirectional link check (BLC), update the sliding window (UPD), forward the OGM (FWD), update and forward (UPD+FWD), or drop it. The result for the static and dynamic topology are very similar. The alternative model drops fewer OGMs (−10%), updates the sliding window more often (+15%), and forwards about the same percentage (28%). However, the alternative model nodes processes overall 10% more OGMs.

A cursory reading of these results suggest that the OGMs that are forwarded, or forwarded and updated by the literal model are the same type of messages that are forwarded and updated by the alternative model. After all, the share of forwarded messages (FWD) and of the forward and updated messages (UPD+FWD) in the literal model adds up to the same share of messages that are forwarded in the alternative model. The following analysis, however, shows that this cursory comparison of how the two models classify OGMs is incorrect.

For a more detailed look how the changes in the alternative model change the way OGMs are classified and processed, we ran the literal model in parallel with

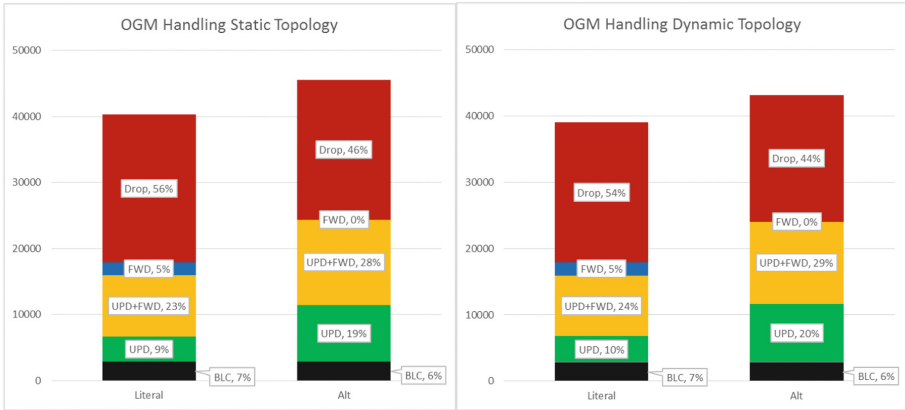


Fig. 7. Classification of received OGMs. Averages over 100 runs.

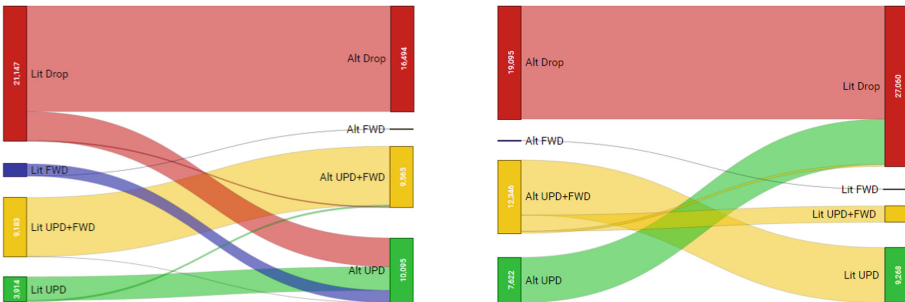


Fig. 8. Classification of OGMs generated in literal model by alternative model rules (left), and vice versa (right).

the classification of the alternative model. The literal model determines which OGMs to forward according to its rules, while the alternative classification logs whether it would have forwarded or dropped the OGM. And vice versa, i.e. the alternative model was run, with the literal model in parallel for logging. Figure 8 compares how OGMs are classified for the dynamic topology.

The analysis shows that the alternative model would use many OGMs that are dropped in the literal model for an update. Furthermore, it would not forward any of the OGMs that the literal model forwards without update. This includes OGMs with an TTL equal to the last TTL. Conversely, the literal model would drop many OGMs that the alternative model uses for an update, and it would not forward most OGMs that the alternative model forwards. This includes the OGMs with a better TTL, and thus a better distance. This cross comparison of the classification of OGMs confirms that the literal and alternative version do forward very different sets of OGMs.

## 5 Conclusion

Formalisation of the B.A.T.M.A.N. protocol revealed several explicit choices that have a negative impact on the route discovery process [2]. This paper proposes an alternative model that departs from the RFC. It includes a sliding window that records the TTL of in-window OGMs. The alternative model then (1) redefines the “best-next-hop” to also include alternative routes, (2) updates the sliding window if better routes are found, (3) forwards only messages that are evidence of *improved routes*, and (4) compares the time-to-live only of messages that have the same sequence number.

These changes allow to reformulate the rules and updates in terms of the TTL of an OGM, i.e. in terms of the distance travelled. The paper presented simulation results for Uppaal models that demonstrate that these changes lead to better route discovery and improved error recovery. This comes at an expense of some additional overhead. The alternative model processes in our simulation experiment about 10% more OGMs.

Future work should investigate how the proposed alternative by Cigno and Furlan [7] could be combined with our alternative model, and how to add these to the latest implementations of B.A.T.M.A.N. This would enable simulation on a classical network simulator and, of course, on a real test bed.

This paper could not present the model in detail. For a more thorough discussion of the formalisation of B.A.T.M.A.N. see [2]. A copy of the models can be downloaded from <http://wwwhome.ewi.utwente.nl/~fehnkera/S17/>.

## References

1. Bulychev, P., David, A., Larsen, K.G., Mikučionis, M., Poulsen, D.B., Legay, A., Wang, Z.: UPPAAL-SMC: statistical model checking for priced timed automata. In: Proceedings 10th Workshop on Quantitative Aspects of Programming Languages and Systems, EPTCS (2012). <https://doi.org/10.4204/EPTCS.85.1>
2. Chaudhary, K., Fehnker, A., Mehta, V.: Modelling, verification, and comparative performance analysis of the B.A.T.M.A.N. protocol. In: Models for Formal Analysis of Real Systems (MARS 2017). EPTCS (2017). <https://doi.org/10.4204/EPTCS.244.3>
3. Clausen, T., Jacquet, P.: Optimized Link State Routing Protocol (OLSR). Network Working Group. <http://www.tools.ietf.org/html/rfc3626>
4. 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](https://doi.org/10.1007/978-3-642-28756-5_13)
5. Fehnker, A., Höfner, P., Kamali, M., Mehta, V.: Topology-based mobility models for wireless networks. In: Joshi, K., Siegle, M., Stoelinga, M., D’Argenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 389–404. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-40196-1\\_32](https://doi.org/10.1007/978-3-642-40196-1_32)
6. Furlan, D.: Analysis of the overhead of B.A.T.M.A.N. routing protocol in regular torus topologies. Technical report, University of Trento, Italy (2011). <https://downloads.open-mesh.org/batman/papers/OGMoverhead.pdf>

7. Furlan, D.: Improving BATMAN routing stability and performance. Master's thesis, University of Trento (2011). [https://downloads.open-mesh.org/batman/papers/Improving BATMAN Routing Stability and Performance.pdf](https://downloads.open-mesh.org/batman/papers/Improving%20BATMAN%20Routing%20Stability%20and%20Performance.pdf)
8. Hardes, T.: Performance analysis and simulation of a Freifunk Mesh network in Paderborn using B.A.T.M.A.N. advanced. Master's thesis, University of Paderborn (2015). <http://thardes.de/wp-content/uploads/2016/03/thesis.pdf>
9. Huhtonen, A.: Comparing AODV and OLSR routing protocols (2004). <http://www.tml.tkk.fi/Studies/T-110.551/2004/papers/Huhtonen.pdf>
10. Kamali, M., Höfner, P., Kamali, M., Petre, L.: Formal analysis of proactive, distributed routing. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 175–189. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-22969-0\\_13](https://doi.org/10.1007/978-3-319-22969-0_13)
11. Kulla, E., Hiyama, M., Ikeda, M., Barolli, L.: Performance comparison of OLSR and BATMAN routing protocols by a MANET testbed in stairs environment. *Comput. Math. Appl.* **63**(2), 339–349 (2012)
12. Marinis Artelaris, S.: Performance evaluation of routing protocols for wireless mesh networks (2016). <http://lnu.diva-portal.org/smash/get/diva2:903013/FULLTEXT01.pdf>
13. Neumann, A., Aichele, C., Lindner, M., Wunderlich, S.: Better approach to mobile ad-hoc networking (B.A.T.M.A.N.). IETF Draft (2008). <https://tools.ietf.org/html/draft-wunderlich-openmesh-manet-routing-00>
14. Wang, J.C.P., Hagelstein, B., Abolhasan, M.: Experimental evaluation of IEEE 802.11s path selection protocols in a mesh testbed. In: 2010 4th International Conference on Signal Processing and Communication Systems (2010). <https://doi.org/10.1109/ICSPCS.2010.5709664>
15. Wibling, O., Parrow, J., Pears, A.: Automatized verification of ad hoc routing protocols. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 343–358. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-30232-2\\_22](https://doi.org/10.1007/978-3-540-30232-2_22)



# Towards a Formal Safety Framework for Trajectories

Marco A. Feliú<sup>(✉)</sup> and Mariano M. Moscato

National Institute of Aerospace, Hampton, USA  
{marco.feliu,mariano.moscato}@nianet.org

**Abstract.** Trajectory generation is at the heart of an autonomous Unmanned Aerial Vehicle (UAV). Given a navigation context, the UAV has to conceive a trajectory that fulfills its mission goal while avoiding collisions with obstacles and surrounding traffic. This *intended trajectory* is an idealization of the various actual *physical trajectories* that the UAV may perform during flight. The validation of actual *physical trajectories* with respect to their *intended* counterparts is challenging due to the interaction over time of several uncontrolled factors such as the environmental conditions, measurement errors and the cyber-physical components of the UAV. For this reason, the most common validation technique for trajectory generators is flight simulation, which is not exhaustive and thus cannot prove the actual absence of collisions.

This paper presents a preliminary formal framework to reason about the safety of UAV trajectories with respect to static-obstacle collision avoidance taking account of the uncertainties derived from uncontrolled factors. The proposed framework was formally verified in a mechanical theorem prover. Its application as an *oracle* for *black-box* testing validation of trajectory generators is also proposed. Such testing strategy would allow the safety evaluation of trajectories while removing the need for simulation procedures, thus reducing the cost of the validation process.

## 1 Introduction

Cyber-physical systems and, in particular, autonomous Unmanned Aerial Vehicle (UAV) are especially hard to analyze. This is because their behavior depends not only on the (correct) interaction of digital and analog components intended to control the system but also on uncontrolled factors. These factors can be either external, for example, the weather conditions affecting the operation of a UAV, as well as internal, such as the uncertainty inherent to the physical aspect of the actuators of a vehicle.

The more widespread evaluation procedures for this kind of systems are simulation and flight testing. Flight testing provides actual information about the real behaviour of the system as a whole. But this information is limited to flying contexts that can be artificially reproduced and it is obtained at a great cost. This is the reason why simulation is the most widely used approach to validation of cyber-physical systems since it can approximate real behavior under any

specific condition at a significative lower cost and without risks. However, none of these techniques is exhaustive in the sense that they cannot prove properties about the behavior of a UAV.

There are two reasons causing the lack of exhaustiveness of simulation and flight testing. One reason is the set of inputs of the systems that will be considered during the validation procedure; since the number of combinations of missions and navigation contexts of a UAV is potentially infinite, it is impossible to reproduce and check all those scenarios. The other reason is the implicit uncertainty of the system performance and the environment in a real setting of the UAV. Since navigation conditions can change instantly during flight and the system itself can behave imprecisely in terms of timing, measurements or actions, it is impossible to reproduce and check all these uncertainty scenarios either. The present work aims to undertake this second aspect of the completeness of the validation process.

This paper introduces a formal framework for reasoning about the collision safety of UAV trajectories in the presence of uncertainty. This framework allows reasoning about all the possible scenarios induced by the uncertainty of the environment and the vehicle when the system is following an *intended trajectory* in a given geographical setting. It does so by overapproximating all the uncertainty scenarios using the formal concept of a *tube* centered in the *intended trajectory*. The proposed framework is being developed using the Prototype Verification System (PVS) [5]. In addition, this paper explains how to apply this framework to the collision-safety validation of a UAV trajectory generator by means of *black-box testing* [2]. To this end, an application of the framework to the concept of a *formally verified safety checker* that can be used as an oracle during the testing process is proposed.

## 2 The Framework

The proposed framework is built on some fundamental concepts that are progressively introduced in this section.

**Trajectories, Control, and Uncertainty.** The framework proposed in this paper allows one to reason about all the possible scenarios induced by the uncertainty of the environment and the vehicle when the system is performing a trajectory. In order to do so, it has to link two related concepts, namely *intended trajectories* and *physical trajectories*, by means of the notions of *control* and *uncertainty*.

**Definition 1 (Intended Trajectory).** *An intended trajectory is the representation of a trajectory that the UAV has planned to perform in order to fulfill its mission or to avoid a collision.*

For example, suppose that a given mission consists in following a flight plan with two waypoints A and B. In this work, the term *flight plan* is understood in



its higher-level meaning: an ordered succession of spacial points, usually called *waypoints*. When the UAV arrives at point A, it may compute an *intended trajectory* in the form of a straight line from A to B. An *intended trajectory* is ideal in the sense that the path by it described might be impossible to be strictly followed by the actual UAV. This difficulty arises from the physical limitations of the vehicle and the uncertainty provoked by its interaction with the environment. Then, such ideal trajectory acts indeed as a reference.

**Definition 2 (Physical Trajectory).** *A physical trajectory is an actual trajectory that a UAV can perform in the real world when it is following an intended trajectory.*

*Physical trajectories* depend on various factors, namely:

- **the control of the vehicle**, depending on the actions taken by the control, the *physical trajectory* will be closer or further from the *ideal* one;
- **the maneuverability of the vehicle**, the shape of the *physical trajectory* will vary according to the kind of maneuvers that the vehicle can perform;
- **the uncertainty of the vehicle behavior**, the vehicle may make wrong measurements of the environment or perform imprecise maneuvers, thus acting slightly different from what is expected;
- **the uncertainty of the environment**, wind or other atmospheric conditions may vary the *physical trajectory* described by the vehicle.

The proposed framework formalizes the relationship between intended and physical trajectories by means of the notion of *control* of the UAV. This *control* works by commanding maneuvers at a given rate in order to allow the vehicle to follow an *intended trajectory* while describing a *physical trajectory* that does not deviate unnecessarily from the *intended* one.

**Definition 3 (Control).** *A function  $Control_{\Delta t}(tr^I, p)$  represents the control of the vehicle. It specifies the actions to be applied by the vehicle actuators in order to move the aircraft from  $p$ , the current position of the vehicle, to the next waypoint according to the intended trajectory  $tr^I$ , assuming that the control will be able to perform the next action at a rate determined by a lapse  $\Delta t$ .*

This framework aims at over approximating all possible physical trajectories for a given intended trajectory. In order to define a complete set of possible *physical trajectories* induced by a *control*, it is necessary to bound the uncertainty of the UAV components and its environment. Let  $U$  represent this bound.

**Definition 4 (AllPhysicalTrajectories).** *Given a function  $Control_{\Delta t}$ , the function  $AllPhysicalTrajectories_{Control_{\Delta t}}^U(tr^I)$  denotes the collection of all possible trajectories resulting from the application of actions decided by the control every time steps not greater than  $\Delta t$  units of time under the uncertainty  $U$ .*

The over approximation of all the potential movement of the UAV is defined as the maximum distance from the *intended trajectory* at which all the *physical trajectories* can be.

**Definition 5 (FollowsAtMaximumDistance).** Given a function  $Control_{\Delta t}$ , the predicate  $FollowsAtMaximumDistance_{Control_{\Delta t}}^U(tr^I, d)$  checks that the control function is able to maintain the vehicle at most at a distance  $d$  of the intended trajectory  $tr^I$  under the uncertainty  $U$ .

As a consequence, if a *control* fulfills the *FollowsAtMaximumDistance* property for a given distance, an over approximation can be built as a *tube*.

**Definition 6 (Tube).** Given an intended trajectory  $tr$  and a distance  $d$ , a *tube*( $tr, d$ ) is the set of points in space whose distance to  $tr$  is no larger than  $d$ .

As shown below, the obstacle-collision-avoidance safety of the *tube* implies the obstacle-collision-avoidance safety of all *physical trajectories*.

**Obstacle-Collision-Avoidance Safety.** The proposed framework has been developed to ensure the *obstacle-collision-avoidance safety* of UAV trajectories by formalizing the notions of *obstacle* and *avoidance*:

**Definition 7 (Obstacle).** An obstacle  $O$  is any geometrical volume.

**Definition 8 (Avoids).** Given a trajectory  $tr$  and a set of obstacles  $Os$ , the predicate  $Avoids(tr, Os)$  states that the trajectory  $tr$  does not collide with any obstacle in  $Os$ .

Since this framework aims at reasoning with over approximations of *physical trajectories*, a notion of *intersection with obstacles* is needed for them as well.

**Definition 9 (Intersects).** Given a tube  $t$  and set of obstacles  $Os$ , the predicate  $Intersects(t, Os)$  states that the tube intersects some obstacle in  $Os$ .

Using the presented notions, it is possible to reason about the obstacle-collision avoidance of physical trajectories from an over approximation.

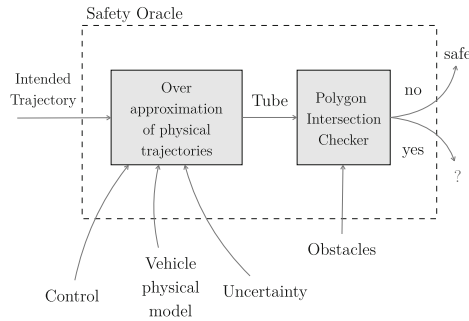
**Theorem 1 (Intended Trajectory Collision Safety).** Given an intended trajectory  $tr^I$ , a control  $Control_{\Delta t}$ , and a set of obstacles  $Os$ ,

$$\begin{aligned} & ( \exists d : FollowsAtMaximumDistance_{Control_{\Delta t}}^U(tr^I, d) \\ & \quad \wedge \neg Intersects(tube(tr^I, d), Os) ) \\ & \Rightarrow \\ & ( \forall tr^P : tr^P \in AllPhysicalTrajectories_{Control_{\Delta t}}^U(tr^I) \\ & \quad \Rightarrow Avoids(tr^P, Os) ) \end{aligned}$$

This theorem reduces the obstacle-avoidance problem to the geometric problem of an intersection of shapes. The *intended trajectory* is considered safe if there exists a *tube*, enclosing all its possible *physical trajectories*, that does not intersect with any of the predefined obstacles. This theorem is the basis for the trajectory validation strategy presented in the next section.

### 3 Trajectory Validation Strategy

The trajectory validation strategy is schematized in Fig. 1. The idea is to have a *safety oracle* that can decide on the obstacle-collision-avoidance safety of *intended trajectories*. This *oracle* first computes an over approximation of all the possible *physical trajectories* that could be generated by a UAV while following the *intended trajectory* to be checked, assuming a specific *control* and *uncertainty* bounds. Then, the *oracle* checks if the over approximation intersects with any of the *obstacles*. If the over approximation does not intersect any *obstacle*, the *oracle* can ensure that the *intended trajectory* is safe. Otherwise, the *oracle* is not able to assure the safety of the input trajectory.



**Fig. 1.** Schematic diagram of the validation strategy.

Theorem 1 supports the use of the *oracle* in this strategy. If an *oracle* is verified to compute approximations satisfying the antecedent of this theorem, its *soundness* as a validator is guaranteed. However, a useful *oracle* for validation must have two additional properties: effectiveness and precision. A practical *oracle* should be *effective* in the sense of being able to decide the safety of a given trajectory consuming a limited and reasonable amount of time and memory space. Moreover, a useful *oracle* will be *precise* in the sense of computing the smallest possible over approximations, so that it can decide on a greater number of *intended trajectories*.

Currently, a simplified kinematic model for physics and control has been formalized. In this model, *intended trajectories* are represented with lists of points, while *physical trajectories* are functions of time into points. In addition, *physical trajectories* are allowed to change their velocity instantaneously, representing the uncertainty of the interaction between the UAV and its environment. Finally, in this model, the *control* assumes uncertainty in its inputs (position and speed) when choosing a movement action.

## 4 Conclusion and Future Work

This paper reports an ongoing effort on the development of a formal framework to reason about the collision safety of trajectories with uncertainty, and its application to a validation strategy for trajectories.

The framework currently relies on some properties of the UAV *control* and a coarse-grained notion of *uncertainty*. Hence, a limitation of the current framework is the need of two formal descriptions: the *control* of the UAV and the *uncertainty* of the environmental conditions and their interaction with the aircraft. In the future, it is planned to focus on these limitations. For the *control*, it is planned to formalize a scheme of over approximations of the *control* itself, to provide a cost-effective way of reasoning on *real-life* examples of such components. The model of *uncertainty* will be refined in order to account for more low-level details and interactions between them and the context of the UAV. Another future improvement is the more realistic description of the *physical* behavior of the UAV. This will produce more precise over approximations making the validation strategy more effective. Furthermore, the ideas behind this framework are compatible with existing approaches to formally represent trajectories or to reason about them, such as [1, 4], thus opening future improvement opportunities.

The framework does not consider traffic-collision safety in its current state. The integration of this or other avoidance-related properties into the framework is also future work. The intuition is that *tubes* may be computed for traffic vehicles and, together with the ownship *tube*, they can be checked for intersection in a 4-dimensional space.



This formal framework is being developed for the validation of the trajectory generator module of the High-Integrity version of the Safe Autonomy Flexible Innovation Testbed (SAFIT<sup>TM</sup>) [3]. The preliminary specification of the framework has been implemented in PVS [5]. The full validation strategy implementation is an ongoing work.

## References

1. Hagen, G., Guerreiro, N.M., Maddalon, J.M., Butler, R.W.: An efficient universal trajectory language. Technical report TM-2017-219669, NASA (2017)
2. Howden, W.E.: Introduction to the theory of testing. In: Tutorial: Software Testing and Validation Techniques, pp. 16–19 (1978)
3. Johnson, S.C., Petzen, A., Tokotch, D.: Exploration of detect-and-avoid and well-clear requirements for small UAS maneuvering in an urban environment. In: American Institute of Aeronautics and Astronautics, 28 November 2017 (2017). <https://doi.org/10.2514/6.2017-3074>
4. Narkawicz, A., Muñoz, C.: Formal verification of conflict detection algorithms for arbitrary trajectories. *Reliab. Comput.* **17**, 209–237 (2012)
5. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992). [https://doi.org/10.1007/3-540-55602-8\\_217](https://doi.org/10.1007/3-540-55602-8_217)



# Static Value Analysis of Python Programs by Abstract Interpretation

Aymeric Fromherz<sup>1,2</sup>, Abdelraouf Ouadjaout<sup>2</sup> , and Antoine Miné<sup>2</sup> 

<sup>1</sup> Carnegie Mellon University, Pittsburgh, USA  
afromher@andrew.cmu.edu

<sup>2</sup> Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6,  
75005 Paris, France  
{abdelraouf.ouadjaout,antoine.mine}@lip6.fr

**Abstract.** We propose a static analysis by abstract interpretation for a significant subset of Python to infer variable values, run-time errors, and uncaught exceptions. Python is a high-level language with dynamic typing, a class-based object system, complex control structures such as generators, and a large library of builtin objects. This makes static reasoning on Python programs challenging. The control flow is highly dependent on the type of values, which we thus infer accurately.

As Python lacks a formal specification, we first present a concrete collecting semantics of reachable program states. We then propose a non-relational flow-sensitive type and value analysis based on simple abstract domains for each type, and handle non-local control such as exceptions through continuations. We show how to infer relational numeric invariants by leveraging the type information we gather. Finally, we propose a relational abstraction of generators to count the number of available elements and prove that no *StopIteration* exception is raised.

Our prototype implementation is heavily in development; it does not support some Python features, such as recursion nor the *compile* builtin, and it handles only a small part of the builtin objects and standard library. Nevertheless, we are able to present preliminary experimental results on analyzing actual, if small, Python code from a benchmarking application and a regression test suite.

## 1 Introduction

Sound static analyzers based on abstract interpretation [7] have been successful in formally checking correctness properties of programs. Academic and industrial successes include, for instance, Polyspace Verifier, Astrée [5], Sparrow [18], and Julia [26]. The major part of these analyzers target solely statically typed languages, such as C, Java, or C#. With the rise of web applications, the static analysis of JavaScript programs has started to gain some attention [2, 14, 15]. The more dynamic nature of the language makes this task challenging. In this

---

This work is partially supported by the European Research Council under Consolidator Grant Agreement 681393 – MOPSA.

article, we look at another dynamic language, Python [21], that, we feel, has been largely neglected by the static analysis community.

Python is a relatively recent programming language, introduced in 1991, which has gained a lot of popularity due to its readable syntax, ease of programming, interactive toplevel, and large library support. It is used notably in education and science, for beginners and non-computer scientists, as a scripting and prototyping language. It is an interpreted language with dynamic features, including dynamic typing (variables are not typed, and can hold values of any type), a class system supporting object run-time alteration (adding fields beyond what is declared in the object class, and possibly altering the class hierarchy), overloading for methods but also builtin language operators (such as +), reflection, closures, and an *eval* keyword. While these dynamic features are a popular aspect of the language, and are effectively relied on in Python programs [1], they make reasoning on Python programs and ensuring the absence of run-time errors very difficult at compile-time. This has motivated the design of Python subsets and variants with more static typing [3], but that does not help with the large majority of existing Python code that does not obey these restrictions.

We design instead a specific analysis for Python that embraces fully the dynamic aspects of the language—we nevertheless draw the line and reject programs featuring code generation, calling *eval* or *compile* builtins, or importing modules from locations that are not statically known. Our abstract interpreter infers the possible values of program variables in a flow-sensitive way. This information allows us to derive the possible types of each variable at each program point, and hence deduce the control flow for the next instruction. Our analysis then detects soundly all possible run-time errors, that is, uncaught exceptions.

*Formal semantics.* Following the standard abstract interpretation road-map, we define a concrete collecting semantics, and then derive an effective analyzer by abstraction. An additional difficulty of Python is the lack of formal specification—unlike, for instance, JavaScript, that features an English specification [9] that provides a sound basis for formal specifications [6]. The Python language is defined by its reference manual [21], which leaves room for ambiguity and permits implementation freedom. We base our own semantics on earlier formalization efforts [20], on the reference manual [21], and on the CPython reference implementation. We innovate by defining the semantics as an input-output function on environments, by induction on the syntax with explicit fixpoints for loops, which lends itself well to the design of an abstract interpreter.

*Value and type analysis.* The core component of our analyzer employs non-relational abstractions, assigning an abstract set of values to each variable. Following the JavaScript analysis by Jensen et al. [14], each variable is given a tuple of abstract values to account for values of all possible types. We employ standard numeric domains, as well as field-sensitive representations for objects abstracted by allocation site, and simple abstractions of builtin Python types (e.g., strings are represented as finite concrete sets or  $\top$ ; lists are represented as a summary object and a length information, etc.). Consider, for instance, that the function `init` in Fig. 1a is called with a function object argument `f` and an

<pre> 1 def init(f, n=None): 2   if n is None or n &lt;= 0 : return [] 3   l = [] 4   for i in range(n): 5     l.append(f(i)) 6   assert len(l) == n </pre>	<pre> 1 def gen(): 2   for i in range(0,10): 3     yield i 4   b = gen() 5   for j in range(0,5): 6     a = next(b) # no StopIteration </pre>
---	---

(a) Dynamic typing example.

(b) Generator example.

**Fig. 1.** Python programs illustrating challenging static analysis situations.

optional argument `n` with a default `None` value. Our value analysis will infer that `n` is never `None` when `range(n)` is evaluated, so that no exception is raised at this point. Additionally, attribute, method, and operator resolution is handled easily by extracting type information from the value abstraction. Another complication we handle is that attributes and methods can be added dynamically to an object beyond what is statically declared in its class.

*Relational numeric analysis.* Additionally, we show how we can go beyond non-relational abstractions and leverage numeric relational domains, such as polyhedra [8], which are invaluable to program analysis—notably to infer non-trivial inductive loop invariants. We rely on a reduction with the non-relational domains to deduce variables that are purely numeric at each program point and can thus be fed to a relational domain. When applying our relational analysis on the previous example shown in Fig. 1a, we are able to prove the assertion at line 6, while the non-relational value analysis will raise a false alarm.

*Generators.* A unique characteristic of Python is the pervasive use of generators, a limited form of co-routines that permeate the standard library. The example in Fig. 1b creates a generator `gen` that returns a new value in  $0, 1, \dots, 9$  at each call to `next`. More precisely, each call to `next` resumes the execution of the iterator, until it calls `yield` and the control is returned to the caller, until the next call to `next`, etc. We develop specific abstractions to model generators, and use a continuation-based iterator to analyze complex, non-local inter-procedural control in an abstract interpreter by induction on the syntax. Combined with relational invariants, the analyzer is able to prove that there are less calls to `next` than to `yield`, so that a `StopIteration` exception is never raised.

*Implementation.* We have implemented a prototype analyzer and run it on a small set of Python benchmarks. The output of the analysis is a superset of all the possible variable values at each program point as well as the set of uncaught exceptions. Note that Python is a large language with many builtin types, primitives, and standard support libraries. We currently support a selected representative set of primitives, that are sufficient to analyze our benchmarks.

*Focus and limitations.* Although we believe that our design is sound and scalable, it currently employs some very naive abstractions with respect to the state of

the art. Our almost-concrete string abstraction could be replaced with the complex abstractions designed by Amadini et al. for JavaScript [2]. Likewise, object abstractions have been studied extensively, especially for the analysis of Java, and we could replace our simple allocation-site abstraction based on recency abstraction [4] with more efficient ones, such as object-sensitive abstractions [24].

Python instructions involving dynamic code generation or retrieval, including *eval* and *compile*, are not supported—although existing work on JavaScript [13] could help. Likewise, we do not support recursive procedures, which are not much employed in Python—classic interprocedural analysis techniques [23] could also apply. Integrating and evaluating these previous works in the context of Python analysis is left as future work. We chose instead to focus our research on novel aspects of the analysis of Python: the integration of relational abstract domains, and the support for generators, which were not considered in previous works.

Finally, the currently scarce support for Python builtins and libraries severely limits the practical usability of our prototype on realistic Python code. We are more interested at the moment in developing relational analyses that go beyond, in term of expressiveness, current analyses for dynamic languages, than supporting imprecisely the entirety of the language primitives. We also note that, to our knowledge, none of the proposed formal semantics of Python [11, 19, 20, 22] were mature enough to analyze actual Python programs without rewriting them, while we are at least able to analyze small benchmarks and tests unmodified.

*Organization.* The rest of the article is organized as follows: Sect. 2 presents the syntax and concrete collecting semantics of our normalized Python subset; Sect. 3 presents a non-relational analysis based on replacing the concrete domain with abstract value domains, as well as a relational abstraction; Sect. 4 presents our generator analysis; Sect. 5 presents our implementation and experimental results. Finally, Sect. 6 discusses related work and Sect. 7 concludes.

## 2 The Mini-Python Language

The language we analyze is a significant subset of Python 3.6, using a simplified syntax removing redundant constructions and syntactic sugar, that we call Mini-Python. Some features that are supported by our implementation are not described here for simplicity: slices, `for` loops, and `import` directives. Some other omitted features are not supported at the moment: *eval* and *compile*, recursion, coroutines (although we do support generators).

### 2.1 Syntax

Following the Python language reference [21], we distinguish between expressions, that return a value, and statements, that do not.

*Expressions.* Expressions, presented in Fig. 2, include constants of various types: integers (in  $\mathbb{Z}$ ), booleans (`True`, `False`), strings. `None` and `NotImpl` are types with



$expr ::= \mathbf{True} \mid \mathbf{False} \mid i \in \mathbb{Z} \mid s \in string$		<i>(constants)</i>	
$\mathbf{None} \mid \mathbf{NotImpl} \mid \mathbf{Undef}$		<i>(singletons)</i>	
$(expr, \dots, expr)$	<i>(tuples)</i>	$expr.string$	<i>(attributes)</i>
$id$	<i>(identifier)</i>	$expr \circ expr$	<i>(binary op.)</i>
$\diamond expr$	<i>(unary op.)</i>	$expr[expr]$	<i>(subscript)</i>
$expr(expr, \dots, expr)$	<i>(call)</i>	$\mathbf{next} expr$	<i>(generator next)</i>

**Fig. 2.** Mini-Python expressions.

$stat ::= expr$	<i>(evaluation)</i>	$id \leftarrow expr$	<i>(assignment)</i>
$id.string \leftarrow expr$	<i>(attribute set)</i>	$\mathbf{return} expr$	<i>(return)</i>
$\mathbf{break}$	<i>(loop exit)</i>	$\mathbf{continue}$	<i>(go to loop head)</i>
$\mathbf{raise} expr$	<i>(exception)</i>	$\mathbf{yield}_i expr$	<i>(generator exit)</i>
$stat; stat$	<i>(sequence)</i>	$\mathbf{while}(expr, stat)$	<i>(loop)</i>
$\mathbf{if\_then\_else}(expr, stat, stat)$			<i>(conditional)</i>
$\mathbf{try\_except\_else}(stat, (string \times stat)^*, stat)$			<i>(exception handling)</i>
$\mathbf{fun}(string, string^*, stat)$			<i>(function declaration)</i>
$\mathbf{gen}(string, string^*, stat)$			<i>(generator declaration)</i>
$\mathbf{class}(string, expr^*, stat)$			<i>(class declaration)</i>

**Fig. 3.** Mini-Python statements.

a single inhabitant each, also denoted as `None` and `NotImpl`. They represent respectively the absence of a value and of a special method (such as `__add__`, modeling `+`). `Undef` denotes the value of uninitialized variables. Expressions also include literal tuples  $(e_1, \dots, e_n)$ , object attributes  $e.string$ , identifiers for variables, functions, and classes, an element of a collection  $e_1[e_2]$  and, finally, a call  $e(e_1, \dots, e_n)$  to any callable object: function, generator, class constructor.

*Statements.* Figure 3 presents the syntax of statements. Most are standard: atomic statements such as expression evaluation, assignment  $e_1 \leftarrow e_2$ , attribute update  $e_1.string \leftarrow e_2$ ; control instructions such as `return`  $e$ , `break`, `continue`, tests `if_then_else`( $c, t, e$ ), and loops `while`( $c, b$ ). Exceptions are raised through `raise`  $e$  and caught through `try_except_else`( $e, clauses, else$ ), where  $clauses$  is a list of pairs  $(name, body)$  assigning a body to specific exception classes, and  $else$  to execute when no exception is raised. Generators generate a value using `yieldi`  $e$ , while the next element of a generator is queried with `next`  $o$ , passing the generator object  $o$  as argument— $o.__next__()$  in Python. Each `yieldi`  $e$  statement is subscripted with a unique syntactic token  $i \in \mathbb{N}$ , used later in the semantic state of generator instances to remember to which `yield` instruction we should jump back when `next`  $o$  is called. Finally, `fun`( $name, args, body$ ) declares a function with a name, a list of formal arguments, and a body; `gen`( $name, args, body$ ) declares a generator similarly; and `class`( $name, bases, body$ ) declares a new class with the given name, inheriting from a list of base classes, and with the given body. Such declarations can appear in any statement, possibly nested in conditionals, loops, or other declarations.

Definitions occur at run-time: the act of executing a definition statement creates a new binding in the environment.

There is a unified namespace for variable names, function names, and class names and we assume that all identifiers in the program are unique. We also restrict the language to recursion-free programs. We will be able to encode environments as maps from names to values without ambiguity. Python features unintuitive scoping rules: due to the lack of variable declarations, any assigned variable automatically gets function scope, even if it is used before it is first assigned. We hoist declarations at the function scope level, explicitly assigning them to **Undef**. The analysis is then able to detect **UnboundLocalError** exceptions due to using uninitialized variables, which is a major issue in Python.

## 2.2 Concrete Collecting Semantics

We define a concrete collecting semantics by induction on the syntax, as a function mapping sets of environments to sets of environments. To handle non-local control flow, such as **break** and **return**, we add a continuation layer to environments. The case of generators is more involved; its description is deferred to Sect. 4. As Python is a large language, we only present here the semantics of a selection of statements that we feel illustrate the specific difficulties of Python semantics and our solutions.

*Program environments and values.* We denote as **Id** the (finite) set of identifiers used in the program and as **Addr** an infinite set of memory addresses. As usual, a memory state is a pair  $m = (\epsilon, \Sigma) \in \mathcal{E} \times \mathcal{H}$ , where the environment  $\epsilon \in \mathcal{E}$  is a partial function assigning a value to existing variables, while the heap  $\Sigma \in \mathcal{H}$  maps currently allocated addresses to objects. Values, in **Val**, can be atomic, such as integers, strings or constants, or addresses of objects, which live in **Obj**:

$$\begin{aligned} \mathcal{E} &\stackrel{\text{def}}{=} \mathbf{Id} \rightarrow \mathbf{Val} \\ \mathcal{H} &\stackrel{\text{def}}{=} \mathbf{Addr} \rightarrow \mathbf{Obj} \\ \mathbf{Obj} &\stackrel{\text{def}}{=} \mathit{string} \rightarrow \mathbf{Val} \\ \mathbf{Val} &\stackrel{\text{def}}{=} \mathbb{Z} \cup \mathit{string} \cup \{\mathbf{True}, \mathbf{False}, \mathbf{None}, \mathbf{NotImpl}, \mathbf{Undef}\} \cup \mathbf{Addr} \end{aligned}$$

Objects map (finitely many) attributes to values. Following Python, we model complex values, including lists, functions, generators, classes, and methods, as objects: **List**, **Fun**, **Gen**, **Class**, **Method**  $\subseteq$  **Obj**. Their special semantic properties are derived from the presence of some attributes. For instance: a list  $l \in \mathbf{List}$  has a length  $l.length \in \mathbb{Z}$ . We assume that identifiers are strings, **Id**  $\subseteq$  *string*, which can be exploited to reify environments  $\epsilon \in \mathcal{E}$  as objects:  $\epsilon \in \mathbf{Obj}$ .

*States and continuations.* To implement non-local control-flow in our input-output semantic, we employ continuations: a semantic state contains not only the current memory state  $(\epsilon, \Sigma)$ , but also memory states at previously encountered jump points, that are meant to flow into the current state when encountering the corresponding jump target. This technique has been used, for instance, in Astrée [5], to model **break** and **return** in C. For Python, we consider the following flow

$$\mathbb{E}[\text{id}](f, \epsilon, \Sigma) \stackrel{\text{def}}{=} \begin{array}{l} \text{if } f \neq \text{cur} \text{ then } (f, \epsilon, \Sigma, \text{None}) \text{ else} \\ \text{if } \epsilon(\text{id}) = \text{NotFound} \text{ then } \text{NameError}(f, \epsilon, \Sigma) \text{ else} \\ \text{if } \epsilon(\text{id}) = \text{Undef} \text{ then } \text{UnboundLocalError}(f, \epsilon, \Sigma) \text{ else } (f, \epsilon, \Sigma, \epsilon(\text{id})) \end{array}$$

$$\text{NameError}(f, \epsilon, \Sigma) \stackrel{\text{def}}{=} \begin{array}{l} \text{let } (f_1, \epsilon_1, \Sigma_1, v_1) = \mathbb{E}[\text{NameError}()] (f, \epsilon, \Sigma) \text{ in} \\ \quad (\text{exn}, \epsilon_1[\text{exn\_var} \mapsto v_1], \Sigma_1, \text{None}) \end{array}$$

(and similarly for `UnboundLocalError` and `TypeError`)

$$\mathbb{E}[e_1 + e_2](f, \epsilon, \Sigma) \stackrel{\text{def}}{=} \begin{array}{l} \text{if } f \neq \text{cur} \text{ then } (f, \epsilon, \Sigma, \text{None}) \text{ else} \\ \text{let } (f_1, \epsilon_1, \Sigma_1, v_1) = \mathbb{E}[e_1] (f, \epsilon, \Sigma) \text{ in} \\ \text{if } f_1 \neq \text{cur} \text{ then } (f_1, \epsilon_1, \Sigma_1, v_1) \text{ else} \\ \text{let } (f_2, \epsilon_2, \Sigma_2, v_2) = \mathbb{E}[e_2] (f_1, \epsilon_1, \Sigma_1) \text{ in} \\ \text{if } f_2 \neq \text{cur} \text{ then } (f_2, \epsilon_2, \Sigma_2, v_2) \text{ else} \\ \text{if } \text{has\_field}(v_1, \text{--add--}, \Sigma_2) \text{ then} \\ \quad \text{let } (f_3, \epsilon_3, \Sigma_3, v_3) = \mathbb{E}[v_1.\text{--add--}(v_2)] (f_2, \epsilon_2, \Sigma_2) \text{ in} \\ \quad \text{if } f_3 \neq \text{cur} \text{ then } (f_3, \epsilon_3, \Sigma_3, v_3) \text{ else} \\ \quad \text{if } v_3 = \text{NotImpl} \wedge \text{typeof}(v_1) \neq \text{typeof}(v_2) \text{ then} \\ \quad \quad \text{if } \text{has\_field}(v_2, \text{--radd--}, \Sigma_3) \text{ then} \\ \quad \quad \quad \text{let } (f_4, \epsilon_4, \Sigma_4, v_4) = \mathbb{E}[v_2.\text{--radd--}(v_1)] (f_3, \epsilon_3, \Sigma_3) \text{ in} \\ \quad \quad \quad \text{if } f_4 \neq \text{cur} \text{ then } (f_4, \epsilon_4, \Sigma_4, v_4) \text{ else} \\ \quad \quad \quad \text{if } v_4 = \text{NotImpl} \text{ then } \text{TypeError}(f_4, \epsilon_4, \Sigma_4) \text{ else } (f_4, \epsilon_4, \Sigma_4, v_4) \\ \quad \quad \text{else } \text{TypeError}(f_3, \epsilon_3, \Sigma_3) \\ \quad \text{else if } v_3 = \text{NotImpl} \text{ then } \text{TypeError}(f_3, \epsilon_3, \Sigma_3) \text{ else } (f_3, \epsilon_3, \Sigma_3, v_3) \\ \text{else if } \text{has\_field}(v_2, \text{--radd--}, \Sigma_2) \wedge \text{typeof}(v_1) \neq \text{typeof}(v_2) \text{ then} \\ \quad \text{let } (f_3, \epsilon_3, \Sigma_3, v_3) = \mathbb{E}[v_2.\text{--radd--}(v_1)] (f_2, \epsilon_2, \Sigma_2) \text{ in} \\ \quad \text{if } f_3 \neq \text{cur} \text{ then } (f_3, \epsilon_3, \Sigma_3, v_3) \text{ else} \\ \quad \text{if } v_3 = \text{NotImpl} \text{ then } \text{TypeError}(f_3, \epsilon_3, \Sigma_3) \text{ else } (f_3, \epsilon_3, \Sigma_3, v_3) \\ \text{else } \text{TypeError}(f_2, \epsilon_2, \Sigma_2) \end{array}$$

Fig. 4. Semantics of a few Mini-Python expressions.

tokens:  $\mathcal{F} \stackrel{\text{def}}{=} \{ \text{cur}, \text{ret}, \text{brk}, \text{cont}, \text{exn} \}$ , where *cur* is the current flow, on which most instructions operate; *ret*, *brk*, *cont*, *exn* collect the set of states jumping respectively from a `return`, a `break`, a `continue`, or a `raise` statement to the end of, respectively the current function, loop, loop iteration, or `try` statement. Our semantics manipulates collections of memory states attached to flow tokens. The concrete domain is thus  $\mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{F} \times \mathcal{E} \times \mathcal{H})$ .

*Semantics.* Expressions return a value, but can also have side effects—including changing the control flow in case an exception is raised. The semantics  $\mathbb{E}[e] : \mathcal{P}(\mathcal{F} \times \mathcal{E} \times \mathcal{H}) \rightarrow \mathcal{P}(\mathcal{F} \times \mathcal{E} \times \mathcal{H} \times \mathbf{Val})$  of an expression  $e$  in some states thus returns a set of states with a value attached. Many expressions map each state to a single state and value, in which case we define  $\mathbb{E}[e]$  as a function  $(\mathcal{F} \times \mathcal{E} \times \mathcal{H}) \rightarrow (\mathcal{F} \times \mathcal{E} \times \mathcal{H} \times \mathbf{Val})$  and leave implicit the lifting to sets of states. Figure 4

$$\begin{array}{ll}
\mathbb{S}[\text{if\_then\_else}(e, s_1, s_2)]S \stackrel{\text{def}}{=} & \mathbb{S}[\text{while}(e, c)]S \stackrel{\text{def}}{=} \\
\mathbb{S}[s_1](\text{filter}(e)(S)) \mathbb{S}[s_2](\text{filter}(\neg e)(S)) & \text{let } S_0 = \{ (f, \epsilon, \Sigma) \in S \mid \\
& \quad f \neq \text{brk}, \text{cont} \} \text{ in} \\
\mathbb{S}[\text{break}]S \stackrel{\text{def}}{=} & \text{let } S_1 = \text{filter}(\neg e)(\text{lfp } \text{post}) \text{ in} \\
\{ (f, \epsilon, \Sigma) \in S \mid f \neq \text{cur} \} \cup & \{ (f, \epsilon, \Sigma) \in S \mid f \in \{ \text{brk}, \text{cont} \} \} \cup \\
\{ (\text{brk}, \epsilon, \Sigma) \mid (\text{cur}, \epsilon, \Sigma) \in S \} & \{ (f, \epsilon, \Sigma) \in S_1 \mid f \neq \text{brk}, \text{cont} \} \cup \\
& \{ (\text{cur}, \epsilon, \Sigma) \mid (\text{brk}, \epsilon, \Sigma) \in S_1 \} \\
\text{filter}(e)(S) \stackrel{\text{def}}{=} \cup_{(f, \epsilon, \Sigma) \in S} \text{filter}(e)(f, \epsilon, \Sigma) & \text{post}(T) \stackrel{\text{def}}{=} \\
\text{filter}(e)(f, \epsilon, \Sigma) \stackrel{\text{def}}{=} & \text{let } T_0 = \mathbb{S}[c](\text{filter}(e)(T)) \text{ in} \\
\text{let } (f_1, \epsilon_1, \Sigma_1, v_1) = \mathbb{E}[e](f, \epsilon, \Sigma) \text{ in} & S_0 \cup T_0 \cup \\
\text{if } f_1 \neq \text{cur} \text{ then } \{(f_1, \epsilon_1, \Sigma_1)\} \text{ else} & \{ (\text{cur}, \epsilon, \Sigma) \mid (\text{cont}, \epsilon, \Sigma) \in T_0 \} \\
\text{let } (f_2, \epsilon_2, \Sigma_2, v_2) = \text{is\_true}(f_1, v_1, \epsilon_1, \Sigma_1) \text{ in} & \\
\text{if } f_2 \neq \text{cur} \text{ then } \{(f_2, \epsilon_2, \Sigma_2)\} \text{ else} & \\
\text{if } v_2 = \text{True} \text{ then } \{(f_2, \epsilon_2, \Sigma_2)\} \text{ else } \emptyset & 
\end{array}$$

Fig. 5. Semantics of a few Mini-Python statements.

gives a representative sample of such expression semantics. Expressions are only evaluated for the current flow *cur*, while states attached to other flows “pass through” the evaluation unchanged—they return a `None` value which is not used. The case of identifiers  $\mathbb{E}[id]$  illustrates the generation of an exception when the variable has not been found or not been initialized: `NameError()` is a constructor that allocates a new object of class `NameError` and returns its address, while the helper function `NameError(f,  $\epsilon$ ,  $\Sigma$ )` binds this new object to the special global variable *exn\_var* denoting the currently raised exception, and shifts the flow token to *exn* to instruct the semantics to ignore the effect of instructions on this environment until an `except` statement is encountered; `UnboundLocalError` and `TypeError` behave similarly. The case of the addition `+` is far more complex, and a good illustration of the complexity of the language—most operators are as complex, and yet sufficiently different from one another to defeat attempts to factor their definitions. We start by evaluating the arguments from left to right. We then execute the `__add__` method from the left argument, if it exists—which is detected using `has_field(v, attr,  $\Sigma$ )`. If it does not exist, or if it returns `NotImpl`, we call the `__radd__` method from the right argument. Note the systematic check that the flow token is still *cur*: a change of flow token denotes an exception that causes the operator to abort while returning the latest environment.

We denote as  $\mathbb{S}[s]: \mathcal{P}(\mathcal{F} \times \mathcal{E} \times \mathcal{H}) \rightarrow \mathcal{P}(\mathcal{F} \times \mathcal{E} \times \mathcal{H})$  the semantics of a statement *s*. Using sets of environments allows us to easily chain statements, so that we define  $\mathbb{S}[s_1; s_2] \stackrel{\text{def}}{=} \mathbb{S}[s_2] \circ \mathbb{S}[s_1]$ . Figure 5 gives the semantics of a few statements that illustrate the use of non-local control flow. As usual, a test filters its environments to keep only those satisfying the condition, or its negation, to execute the respective branch, and merges them with a union. Filters use `is_true` (omitted for concision) to compute truth values; similarly to `+`, it successively tries to call the special methods `__bool__` and `__len__`, and returns `True` if none of these methods are implemented. The semantics of loops computes, as usual, a least fixpoint. Its definition is complicated by non-local control: a

`break` instruction shifts the current environment into a *brk* continuation, which is consumed by the loop semantics to compute the actual exiting environments. The case of `continue` and `return`, as well as exception handling, is similar.

### 3 Value Abstraction

We now present our static analysis of Python. Following the Abstract Interpretation framework, it is designed by abstraction of the concrete semantics from the previous section. The result is an interpreter by induction on the syntax following closely the concrete semantics, using standard non-relational and relational domains, and modeling control flow through partitioning by flow tokens.

#### 3.1 Non-relational Abstraction

We first consider non-relational abstractions: each variable is assigned an abstract value in  $\mathbf{Val}^\sharp$  representing a set of possible concrete values. Following [14], we abstract separately each type of values in its abstract domain, while their product  $\mathbf{Val}^\sharp$  can represent sets of heterogeneous values:

$$\mathbf{Val}^\sharp \stackrel{\text{def}}{=} \mathbf{Undef}^\sharp \times \mathbf{None}^\sharp \times \mathbf{NotImpl}^\sharp \times \mathbf{Bool}^\sharp \times \mathbf{Num}^\sharp \times \mathbf{String}^\sharp \times \mathcal{P}(\mathbf{Addr}^\sharp)$$

For finite types, each domain tracks the presence of each possible value. For instance,  $\mathbf{Undef}^\sharp \stackrel{\text{def}}{=} \{\perp, \mathbf{Undef}\}$ , where  $\perp$  denotes the definite absence of `Undef`, while `Undef` denotes the possible presence of `Undef`;  $\mathbf{None}^\sharp$  and  $\mathbf{NotImpl}^\sharp$  are similar, while  $\mathbf{Bool}^\sharp \stackrel{\text{def}}{=} \{\perp, \mathbf{True}, \mathbf{False}, \top\}$ . Our string domain is simply the finite sets of strings, plus a  $\top$  element to denote any string:  $\mathbf{String}^\sharp \stackrel{\text{def}}{=} \mathcal{P}_{\text{finite}}(\text{string}) \cup \{\top\}$ . More clever abstractions, such as [2], will be considered in future work. We can use any non-relational domain for  $\mathbf{Num}^\sharp$ , and our implementation uses integer and float intervals. To finitely represent the heap, we use a classic allocation-site abstraction of `Addr` into a finite set  $\mathbf{Addr}^\sharp$  of abstract addresses—our implementation actually uses recency abstraction [4], which we omit in our formalization for simplicity. An abstract tuple  $V = (V_{\mathbf{Undef}}, \dots, V_{\mathbf{String}}, V_{\mathbf{Addr}}) \in \mathbf{Val}^\sharp$  then represents the join of elements from the type-based abstractions:

$$\gamma_{\mathbf{Val}}(V) \stackrel{\text{def}}{=} \gamma_{\mathbf{Undef}}(V_{\mathbf{Undef}}) \cup \dots \cup \gamma_{\mathbf{String}}(V_{\mathbf{String}}) \cup (\cup_{a \in V_{\mathbf{Addr}}} \gamma_{\mathbf{Addr}}(a))$$

The definition of the join  $\cup_{\mathbf{Val}}^\sharp$ , subset  $\subseteq_{\mathbf{Val}}^\sharp$ , and widening  $\nabla_{\mathbf{Val}}$  operators on this abstract domain is pointwise and straightforward.

Given abstract values  $\mathbf{Val}^\sharp$  and addresses  $\mathbf{Addr}^\sharp$ , environments  $\epsilon^\sharp$  map variables to values, and stores  $\Sigma^\sharp$  map addresses to objects, as in the concrete:

$$\begin{aligned} \epsilon^\sharp &\in \mathcal{E}^\sharp \stackrel{\text{def}}{=} \mathbf{Id} \rightarrow \mathbf{Val}^\sharp \\ \Sigma^\sharp &\in \mathcal{H}^\sharp \stackrel{\text{def}}{=} \mathbf{Addr}^\sharp \rightarrow \mathbf{Obj}^\sharp \\ \text{where } \mathbf{Obj}^\sharp &\stackrel{\text{def}}{=} (\text{string} \rightarrow \mathbf{Val}^\sharp) \times \mathcal{P}(\text{string}) \end{aligned}$$

$$\begin{aligned}
S^\sharp[\mathbf{break}]S^\sharp &\stackrel{\text{def}}{=} S^\sharp[cur \mapsto \perp_{\mathcal{M}}, brk \mapsto S^\sharp(cur) \cup_{\mathcal{M}}^\sharp S^\sharp(brk)] \\
S^\sharp[\mathbf{while}(e, c)]S^\sharp &\stackrel{\text{def}}{=} \\
&\text{let } S_0^\sharp = S^\sharp[brk, cont \mapsto \perp_{\mathcal{M}}] \text{ in} \\
&\text{let } S_1^\sharp = \text{filter}^\sharp(\neg e)(\lim \lambda T^\sharp. T^\sharp \nabla (S_0^\sharp \cup_{\mathcal{M}}^\sharp \text{post}^\sharp(T^\sharp))) \text{ in} \\
&S_1^\sharp[cur \mapsto S_1^\sharp(cur) \cup_{\mathcal{M}}^\sharp S_1^\sharp(brk), brk \mapsto S^\sharp(brk), cont \mapsto S^\sharp(cont)] \\
&\text{where } \text{post}^\sharp(T^\sharp) \stackrel{\text{def}}{=} (S^\sharp[c] \circ \text{filter}^\sharp(e))(T^\sharp[cur \mapsto T^\sharp(cur) \cup_{\mathcal{M}}^\sharp T^\sharp(cont)])
\end{aligned}$$

**Fig. 6.** Abstract semantics of a few Mini-Python constructions.

Due to address abstraction, an abstract object may represent a set of concrete objects with different attributes. Abstract objects are pairs  $(attr, must) \in \mathbf{Obj}^\sharp$ , where  $attr$  maps all possible object attributes to their values, while  $must$  is the subset of attributes from  $dom(attr)$  that are guaranteed to exist in all objects:

$$\gamma_{\mathbf{Obj}}(attr, must) \stackrel{\text{def}}{=} \{o \in \mathbf{Obj} \mid must \subseteq dom(o) \subseteq dom(attr) \wedge \forall i \in dom(o) : o(i) \in \gamma_{\mathbf{Val}}(attr(i))\}$$

The  $must$  information is important to precisely rule out **AttributeError** exceptions.

Finally, we partition abstract states with respect to flow tokens in  $\mathcal{F}$ . Hence, an abstract element lives in  $\mathcal{D}^\sharp \stackrel{\text{def}}{=} \mathcal{F} \rightarrow (\mathcal{E}^\sharp \times \mathcal{H}^\sharp)$ , with concretization:

$$\begin{aligned}
\gamma(X^\sharp) &\stackrel{\text{def}}{=} \{(f, \epsilon, \Sigma) \mid (\epsilon, \Sigma) \in \gamma_{\mathcal{M}}(X^\sharp(f))\} \\
\text{where } \gamma_{\mathcal{M}}(\epsilon^\sharp, \Sigma^\sharp) &\stackrel{\text{def}}{=} \{(\epsilon, \Sigma) \mid dom(\Sigma) \subseteq (\cup_{a^\sharp \in dom(\Sigma^\sharp)} \gamma_{\mathbf{Addr}}(a^\sharp)) \wedge \\
&\quad \forall i : \epsilon(i) \in \gamma_{\mathbf{Val}}(\epsilon^\sharp(i)) \wedge \\
&\quad a \in \gamma_{\mathbf{Addr}}(a^\sharp) \implies \Sigma(a) \in \gamma_{\mathbf{Obj}}(\Sigma^\sharp(a^\sharp))\}
\end{aligned}$$

The join  $\cup^\sharp$  on abstract states is pointwise. Note that it joins the  $must$  attribute information for objects with an intersection  $\cap$ :

$$\begin{aligned}
X_1^\sharp \cup^\sharp X_2^\sharp &\stackrel{\text{def}}{=} \lambda F \in \mathcal{F}. X_1^\sharp(F) \cup_{\mathcal{M}}^\sharp X_2^\sharp(F) \\
\text{where } (\epsilon_1^\sharp, \Sigma_1^\sharp) \cup_{\mathcal{M}}^\sharp (\epsilon_2^\sharp, \Sigma_2^\sharp) &\stackrel{\text{def}}{=} (\lambda i. \epsilon_1^\sharp(i) \cup_{\mathbf{Val}}^\sharp \epsilon_2^\sharp(i), \lambda a^\sharp. \Sigma_1^\sharp(a^\sharp) \cup_{\mathbf{Obj}}^\sharp \Sigma_2^\sharp(a^\sharp)) \\
\text{and } (a_1, m_1) \cup_{\mathbf{Obj}}^\sharp (a_2, m_2) &\stackrel{\text{def}}{=} (\lambda s. a_1(s) \cup_{\mathbf{Val}}^\sharp a_2(s), m_1 \cap m_2)
\end{aligned}$$

Figure 6 gives the abstract semantics for a few Mini-Python constructions. It is similar to the concrete one, up to the partitioning with respect to flow tokens. For instance, a **break** statement merges with a join  $\cup_{\mathcal{M}}^\sharp$  the current flow with that of the accumulated break flows, and empties the current flow. Similarly, the loop incorporates back the continue flow at the loop head, and the break flow at its end, after which the continue and break flow from any enclosing loop is restored. Additionally, it replaces the least-fixpoint with a limit  $\lim$  of the iteration accelerated with a widening  $\nabla$ , which applies  $\nabla_{\mathbf{Val}}$  pointwise.

### 3.2 Relational Abstraction

We now present how we leverage relational numeric domains in a dynamically typed language to improve the analysis precision. The intuition is to maintain

$$\begin{aligned}
 \mathcal{S}_{\mathcal{R}}^{\#} \llbracket id \leftarrow e \rrbracket S^{\#} &\stackrel{\text{def}}{=} \\
 \text{let } S_0^{\#} &= \text{merge}(S^{\#}, \mathcal{S}_{\mathcal{R}}^{\#} \llbracket id \leftarrow e \rrbracket (\text{extract}(S^{\#}))) \text{ in} \\
 \text{let } ((\epsilon_0^{\#}, \Sigma_0^{\#}), \nu_0^{\#}) &= S_0^{\#}(cur) \text{ in} \\
 \text{let } \nu_1^{\#} &= \text{if is\_num}(id)(\epsilon_0^{\#}) \text{ then } \mathcal{S}_{\mathcal{N}}^{\#} \llbracket id \leftarrow e \rrbracket \nu_0^{\#} \text{ else } \mathcal{S}_{\mathcal{N}}^{\#} \llbracket id \leftarrow \top \rrbracket \nu_0^{\#} \text{ in} \\
 S_0^{\#}[cur \mapsto &((\epsilon_0^{\#}, \Sigma_0^{\#}), \nu_1^{\#})] \\
 \\
 \text{filter}_{\mathcal{R}}^{\#}(e)S^{\#} &\stackrel{\text{def}}{=} \\
 \text{let } S_0^{\#} &= \text{merge}(S^{\#}, \text{filter}_{\mathcal{R}}^{\#}(e)(\text{extract}(S^{\#}))) \text{ in} \\
 \text{let } ((\epsilon_0^{\#}, \Sigma_0^{\#}), \nu_0^{\#}) &= S_0^{\#}(cur) \text{ in} \\
 \text{let } \nu_1^{\#} &= \bigcap_{v \in \text{vars}(e)}^{\#} \text{filter}_{\mathcal{N}}^{\#}(e \wedge \text{inf}(v)(\epsilon_0^{\#}) \leq v \leq \text{sup}(v)(\epsilon_0^{\#}))(\nu_0^{\#}) \text{ in} \\
 S_0^{\#}[cur \mapsto &((\epsilon_0^{\#}, \Sigma_0^{\#}), \nu_1^{\#})]
 \end{aligned}$$

where:

$$\begin{aligned}
 \text{extract}(S^{\#}) &\stackrel{\text{def}}{=} \lambda f. \text{let } ((\epsilon^{\#}, \Sigma^{\#}), \_) = S^{\#}(f) \text{ in } (\epsilon^{\#}, \Sigma^{\#}) \\
 \text{merge}(S_r^{\#}, S_{nr}^{\#}) &\stackrel{\text{def}}{=} \lambda f. \text{let } (\_, \nu^{\#}) = S_r^{\#}(f) \text{ in } (S_{nr}^{\#}(f), \nu^{\#}) \\
 \text{is\_num}(id)(\epsilon^{\#}) &\stackrel{\text{def}}{=} \epsilon^{\#}(id) \subseteq_{\mathbf{Val}}^{\#} (\perp, \perp, \perp, \perp, \top_{num}, \perp, \emptyset) \\
 \text{inf}(id)(\epsilon^{\#}) &\stackrel{\text{def}}{=} \text{if is\_num}(id)(\epsilon^{\#}) \text{ then } \text{inf}(\epsilon^{\#}(id).num) \text{ else } -\infty \\
 \text{sup}(id)(\epsilon^{\#}) &\stackrel{\text{def}}{=} \text{if is\_num}(id)(\epsilon^{\#}) \text{ then } \text{sup}(\epsilon^{\#}(id).num) \text{ else } +\infty
 \end{aligned}$$

**Fig. 7.** Abstract relational semantics of atomic statements.

relations among pure numeric variables only. We exploit, in a reduced product, the type information provided by  $\mathcal{D}^{\#}$  to update the relational invariant dynamically when the type of a variable changes. Let us assume that we are given a numeric abstract domain  $\mathcal{N}^{\#}$ , such as octagons [17] or polyhedra [8], provided with classic operators, such as a concretization  $\gamma_{\mathcal{N}} \in \mathcal{N}^{\#} \rightarrow \mathcal{P}(\mathbf{Id} \rightarrow \mathbb{Z})$ , transfer functions  $\mathcal{S}_{\mathcal{N}}^{\#} \llbracket stmt \rrbracket \in \mathcal{N}^{\#} \rightarrow \mathcal{N}^{\#}$ , and condition filters  $\text{filter}_{\mathcal{N}}^{\#}(e) \in \mathcal{N}^{\#} \rightarrow \mathcal{N}^{\#}$ . We define our relation-aware domain as  $\mathcal{D}_{\mathcal{R}}^{\#} \stackrel{\text{def}}{=} \mathcal{F} \rightarrow ((\mathcal{E}^{\#} \times \mathcal{H}^{\#}) \times \mathcal{N}^{\#})$  with the following concretization:

$$\gamma_{\mathcal{R}}(X^{\#}) \stackrel{\text{def}}{=} \{ (f, \epsilon, \Sigma) \mid \text{let } ((\epsilon^{\#}, \Sigma^{\#}), \nu^{\#}) = X^{\#}(f) \text{ in } (\epsilon, \Sigma) \in \gamma_{\mathcal{M}}(\epsilon^{\#}, \Sigma^{\#}) \wedge \\
 \exists \nu \in \gamma_{\mathcal{N}}(\nu^{\#}), \forall id \in \text{dom}(\nu) : \epsilon(id) = \nu(id) \}$$

The concretization performs a reduction between the relational and non-relational environments. The reduction with the heap objects is similar but it is omitted here for simplicity.

Some transfer functions in  $\mathcal{D}_{\mathcal{R}}^{\#}$  are given in Fig. 7. They show the interaction between  $\mathcal{D}^{\#}$  and  $\mathcal{N}^{\#}$ . After an assignment, the type of the *lhs* variable is checked by the non-relational domain. If its value is necessarily numeric, the statement is also applied in the numeric environment  $\nu^{\#}$ ; otherwise the variable is removed from  $\nu^{\#}$ . When applying a filter, mixed-type variables can be constrained to become pure numeric variables. The pre-condition numeric environment  $\nu_0^{\#}$  has no information on them, and they are thus created and initialized with interval information extracted from the non-relational environment  $\epsilon_0^{\#}$ .

We illustrate these abstractions through our motivating example from Fig. 1a. Assume that the function `init` is called with the abstract environment  $\{(cur, \epsilon^\sharp = \langle n \mapsto (\text{None} \vee [10, 100]), \dots \rangle), \nu^\sharp = \top_{\mathcal{N}}, \Sigma^\sharp)\}$ . In the `else` branch at line 3,  $\epsilon^\sharp$  is filtered and `n` becomes numeric, which allows  $\nu^\sharp$  to be refined with the invariant  $10 \leq n \leq 100$ . An expressive enough domain can then prove  $0 \leq i = \text{len}(1) < n$  at line 5 inside the loop, so that the `assert` statement at line 6 is satisfied.

## 4 Generator Analysis

Generators allow a called function to suspend itself with a `yield` statement, storing its state into an object, and resume its execution later with a `next`. We now show how to leverage our continuation-based semantics to analyze them.

$$\begin{aligned} \mathbb{E}[\text{next } e](f, \epsilon, \Sigma) &\stackrel{\text{def}}{=} \\ &\text{if } f \neq \text{cur} \text{ then } (f, \epsilon, \Sigma, \text{None}) \text{ else} \\ &\text{let } (f_1, \epsilon_1, \Sigma_1, v) = \mathbb{E}[e](f, \epsilon, \Sigma) \text{ in} \\ &\text{if } f_1 \neq \text{cur} \text{ then } (f_1, \epsilon_1, \Sigma_1, v) \text{ else} \\ &\text{if } \Sigma_1(v) = \mathbf{Gen}(\text{cont}, \text{frame}, \text{body}, \text{vars}) \text{ then} \\ &\quad \text{if } \text{cont} = \text{end} \text{ then } \text{StopIteration}(f_1, \epsilon_1, \Sigma_1) \text{ else} \\ &\quad \text{let } S_1 = \text{if } \text{cont} = \text{start} \text{ then } \mathbb{S}[\text{body}](\text{cur}, \epsilon_1 \cup \text{frame}, \Sigma_1) \\ &\quad \quad \quad \text{else } \mathbb{S}[\text{body}](\text{next}(i), \epsilon_1 \cup \text{frame}, \Sigma_1) \text{ in} \\ &\quad \{ (\text{cur}, \epsilon_{|\text{Id} \setminus \text{var}}, \Sigma[v \mapsto \mathbf{Gen}(j, \epsilon_{|\text{var}}, \text{body}, \text{vars})], \epsilon(\text{yield\_var})) \mid \\ &\quad \quad (\text{yield}(j), \epsilon, \Sigma) \in S_1, j \in \mathbb{N} \} \cup \\ &\quad \{ (\text{exn}, \epsilon, \Sigma[v \mapsto \mathbf{Gen}(\text{end}, [], \text{body}, \text{vars})], \text{None}) \mid (\text{exn}, \epsilon, \Sigma) \in S_1 \} \cup \\ &\quad \{ \text{StopIteration}(\text{cur}, \epsilon, \Sigma[v \mapsto \mathbf{Gen}(\text{end}, [], \text{body}, \text{vars})]) \mid \\ &\quad \quad (f, \epsilon, \Sigma) \in S_1 \wedge f \neq \text{exn}, \text{yield} \} \\ &\text{else } \mathbb{E}[v.\text{--next--}()] (f_1, \epsilon_1, \Sigma_1) \\ \mathbb{S}[\text{yield}_i e] S &\stackrel{\text{def}}{=} \\ &\text{let } S_1 = \mathbb{S}[\text{yield\_var} \leftarrow e] S \text{ in} \\ &\{ (\text{cur}, \epsilon, \Sigma) \mid (\text{next}(i), \epsilon, \Sigma) \in S_1 \} \cup \{ (\text{yield}(i), \epsilon, \Sigma) \mid (\text{cur}, \epsilon, \Sigma) \in S_1 \} \cup \\ &\{ (f, \epsilon, \Sigma) \in S_1 \mid f \neq \text{cur}, \text{next}(i) \} \end{aligned}$$

Fig. 8. Concrete semantics of generators in Mini-Python.

### 4.1 Concrete Semantics

We extend flow tokens to represent continuations able to jump between `next` and `yield` instructions:  $\mathcal{F}_g \stackrel{\text{def}}{=} \mathcal{F} \cup \{ \text{next}(i), \text{yield}(i) \mid i \in \mathbb{N} \}$ , where  $i$  represents a syntactic label to identify statements. A generator is an object  $g = \mathbf{Gen}(\text{cont}, \text{frame}, \text{body}, \text{vars}) \in \mathbf{Gen} \subseteq \mathbf{Obj}$  which is given, upon creation, a body to execute and its set  $\text{vars} \subseteq \mathbf{Id}$  of local variables. It also maintains some state information: a map  $\text{frame} \in \text{vars} \rightarrow \mathbf{Val}$  from local variables to



$$\begin{aligned}
 \mathbb{S}^\sharp \llbracket \text{yield}_i \ e \rrbracket S^\sharp &\stackrel{\text{def}}{=} \\
 \text{let } S_1^\sharp &= \mathbb{S}^\sharp \llbracket \text{yield\_var} \leftarrow e \rrbracket S^\sharp \text{ in} \\
 S_1^\sharp[\text{cur} \mapsto S_1^\sharp(\text{next}(i)), \text{yield}(i) \mapsto S_1^\sharp(\text{cur}) \cup_{\mathcal{M}}^\sharp S_1^\sharp(\text{yield}(i))]
 \end{aligned}$$

**Fig. 9.** Abstract semantics of `yield`.

<pre> 1 def gen(): 2   for i in range(0,10): 3     yield i 4   b = gen() 5   for j in range(0,5): 6     a = next(b) # no StopIteration                 </pre> <p>(a) Original generator example</p>	<pre> 1 def gen(): 2   for i in range(0,10): 3     yield i 4   b = gen() 5   %counter = 0 6   for j in range(0,5): 7     %counter += 1 8     a = next(b) # no StopIteration                 </pre> <p>(b) Instrumented generator</p>
---	--

**Fig. 10.** Generator example (a) and its instrumented version with counters (b).

values, stored at `yield` statements and restored at the following `next`, and the location  $cont \in \mathcal{C}$  where to resume execution upon the following `next`, where  $\mathcal{C} \stackrel{\text{def}}{=} \mathbb{N} \cup \{start, end\}$  denotes either the beginning (*start*) of the generator before the first `next`, or a `yieldi` statement ( $i \in \mathbb{N}$ ), or the end (*end*) of the generator—meaning that a call to `next` raises a `StopIteration` exception.

The concrete semantics of `next` and `yield` is given in Fig. 8. Each call to `next` executes the generator body from the beginning but sets the flow token to  $next(i)$ : it instructs the interpreter to ignore the effect of statements until reaching the corresponding `yieldi`, effectively modeling a jump to the correct location. The `yieldi`  $e$  statement uses a  $yield(i)$  flow token to skip remaining statements and return to the calling `next`. A global `yield_var` variable is used to transfer the value of  $e$  to `next`, while  $\epsilon_{|var}$  and  $\epsilon_{|Id \setminus var}$  extract the values of the local variables at `yield` and freeze them into the *frame* attribute of the generator. They are restored into the environment at the following `next`.

## 4.2 Abstractions

*Value abstraction.* The concrete modeling of generators can be reduced to simple kinds of operations: flow token updates, and copies between local variables and entries in the generator *frame*, which can be seen as object attributes. Section 3 showed how to abstract these operators, and it is thus easy to enrich it to support generators without the need to enrich the abstract domains at all. This is illustrated in Fig. 9 for the `yield` statement—the case of `next` is similar.

This abstraction is sufficient to infer valuable information on the type, value, and even numeric relations between the local variables of a generator. However,

it does not always precisely match the flow of control between `yield` and `next` instructions, leading to spurious exceptions. Consider the example in Fig. 10a. An interval analysis with widening will correctly infer that  $i \in [0, 9]$  and  $j \in [0, 4]$ . However, the abstraction states that all calls to `next` can jump back to the `yield` statement, whatever the number of iterations of the loop indexed by `i`. In particular, at iteration 10, the generator exits the loop, causing a (spurious) *StopIteration* exception.

*Counting abstraction.* We solve this precision issue by automatically instrumenting programs to keep track of the number of calls to `next` and `yield` through a counter. We show in Fig. 10b a version with this counter explicit.<sup>1</sup> We maintain the counter in both the frame of the caller and the frame of the generator—which is stored in its *frame* attribute. Using a relational domain, such as octagons, allows the analysis to establish both equalities  $\%counter = i + 1$  at line 3, and  $\%counter = j + 1$  at line 7. As  $j \in [0, 3]$ , we deduce that  $i \in [0, 3]$  as well, i.e., we never actually exit the loop at line 2 and never raise a *StopIteration* exception. Through the counter, relations between a generator and its caller can be established.

## 5 Experimental Evaluation

We have implemented our method in a prototype static analyzer in OCaml and tested it on two categories of benchmarks. Firstly, regression tests from the official Python 3.6.3 distribution were used to assess the correctness of the implementation. Secondly, to evaluate precision and efficiency, we have considered programs from the Python Performance Benchmark Suite,<sup>2</sup> which employ more realistic and challenging constructions. Our analyzer reports all uncaught exceptions: type errors, name errors, unbound locals, stop iterations, failed assertions.

*Regression tests.* The official regression tests suite consists in a large number of test programs (nearly 500) covering the builtins of the language and the standard libraries. Since our prototype supports only a subset of Python builtins, we have selected only the handful of tests that target the implemented features. The results of the analysis are presented in Table 1.

For each program, we compute the analysis time (in milliseconds) and the number of unit test methods that (i) were proven correct, (ii) raised exceptions and assertion violations, (iii) were not completely analyzed due to the presence of unsupported language features. We investigated the failed tests to check whether the alarms are real or spurious. The obtained outcomes for each regression test are shown in columns 5 to 8. The last column gives the percentage of test methods that we were able to analyze completely. No alarm was detected, which argues in favor of our analyzer faithfully modeling the language semantics, as the tests do not raise errors when executed by the Python interpreter either, and

<sup>1</sup> A global variable is used for illustration purposes. In practice, a counter is an attribute attached to the generator instance.

<sup>2</sup> <https://github.com/python/performance>.

**Table 1.** Experimental results on regression tests. Result categories: ✓ test passed with no false alarm, ✗ test failed with no false alarm, ? test failed with false alarms, \* test containing unsupported builtins.

Program	Lines	Analysis time	Tests	✓	✗	?	*	Coverage
<code>test_augassign</code>	273	645 ms	7	4	0	2	1	85.71%
<code>test_baseexception</code>	141	20 ms	10	6	0	0	4	60.00%
<code>test_bool</code>	294	47 ms	26	12	0	0	14	46.15%
<code>test_builtin</code>	454	360 ms	21	3	0	0	18	14.29%
<code>test_contains</code>	77	418 ms	4	1	0	0	3	25.00%
<code>test_int_literal</code>	91	29 ms	6	6	0	0	0	100.00%
<code>test_int</code>	218	88 ms	8	3	0	0	5	37.50%
<code>test_list</code>	106	88 ms	9	3	0	0	6	33.33%
<code>test_unary</code>	39	11 ms	6	2	0	0	4	33.33%

they generally test a single execution. Also, the precision of our prototype analyzer is reflected by the low false alarm rate: only 2 unit tests among 97 resulted in spurious violations of `assert` statements. Finally, due to the incomplete support for builtins, the analyzer was unable to analyze some methods, resulting in low coverage ratios in many cases. However, the analyzer is still under development and features a modular architecture that allows adding missing builtins abstractions easily, without requiring the modification of existing code.

*Relational tests.* We have analyzed three programs from the Python Performance Benchmark Suite: `float`, `fannkuch`, and `nbody` (around 270 lines of Python in total) and we varied the underlying numeric domains to show the impact of relational information on the analysis. Firstly, the analysis of these programs using the interval domain terminated in less than 3s: `float` was proven correct but `fannkuch` and `nbody` resulted in a total of 5 false alarms. Using octagons, the analysis time increased to 10 min 10 s, but the number of false alarms was reduced to 3. Finally, an analysis with the polyhedra domain was able to prove the correctness of both `float` and `fannkuch` under 5 s, but the analysis of `nbody` did not terminate before a timeout of 30 min. The scalability is limited because each relational abstract element currently contains all variables and object attributes; classic packing techniques [5] would help us improve this situation.

## 6 Related Work

Several works aim at restricting Python towards more static typing, as Mypy or RPython [3], to ease program verification. While this would help design future programs, a static analyzer for existing code is still invaluable. Note also that Python features static analyzer tools, such as Pylint that, while helping the user, are not based on a formal semantics and do not attempt to be sound.

While [22] proposed a semantics for an object-free Python, the first realistic formal semantics of Python was proposed by Smeding [25] in 2009 for Python 2.5 in Haskell, followed by [20] for Python 3.2 in Racket, and [11] for Python 3.3 in K, inspired from related work on JavaScript [10]. They present small-step executable operational semantics that aim at being tested against CPython’s own regression tests, although experiments were limited by the lack of support for advanced language features and libraries used by the tests—an issue from which we also suffer. Poli [19] presents the first attempt at deriving an abstract semantics, but remains uninstantiated as no abstract value domains are provided. Hassan [12] proposes a static typing using SMT solvers, but require variables to have a single type in the program, a limitation that we overcome. We provide the first complete and implemented abstract interpreter for (a subset of) Python. Unlike previous works, we opted for a big-step semantics, which maps conveniently to an abstract interpreter by induction on the syntax. Continuations have been employed before in abstract interpreters to model control flow, as in Astrée for C [5]; we go one step further by handling exceptions and generators.

We find the works closest to ours in the abstract interpretation of JavaScript. Our non-relational abstraction resembles that of Jensen et al. [14] and later [15]. We go one step further by leveraging relational abstractions as well, which were absent in previous works up to our knowledge. Certain non-relational domains differ due to the different nature of the language and properties we seek, notably our need to under-approximate sets of strings to precisely detect *AttributeError* exceptions. Nevertheless, we could benefit from more advanced string domains as proposed in [16]. Likewise, the analysis of practical uses of *eval* in JavaScript [13] could be the basis to support the equivalent construction in Python.

## 7 Conclusion

We have presented the first static analysis for a realistic subset of Python, able to infer the types and values of variables, and the exceptions that can be raised. In addition to its novel language target, its main characteristics are the ability to infer numeric relations despite dynamic typing, and the support for generators. Our implementation is currently limited to a small subset of Python builtins and standard libraries; nevertheless, it is sufficient to analyze a few small Python programs from actual tests and benchmarking suites, without modification.

Our prototype is a work in progress. Planned work include completing the support for builtins and libraries to be able to analyze Python applications. We also wish to enrich the abstractions used in our analyzer, targeting in particular abstractions proposed for JavaScript [2, 13, 14] and Java [24]. The static analysis of dynamic languages, and in particular of Python, is still a new field. There is much to do to raise its effectiveness to that of the analysis of static languages, such as C.




## References

1. Åkerblom, B., Stendahl, J., Tumlin, M., Wrigstad, T.: Tracing dynamic features in python programs. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pp. 292–295. ACM (2014)
2. Amadini, R., et al.: Combining string abstract domains for JavaScript analysis: an evaluation. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 41–57. Springer, Heidelberg (2017). [https://doi.org/10.1007/978-3-662-54577-5\\_3](https://doi.org/10.1007/978-3-662-54577-5_3)
3. Ancona, D., Ancona, M., Cuni, A., Matsakis, N.D.: RPython: a step towards reconciling dynamically and statically typed OO languages. In: Proceedings of the 2007 Symposium on Dynamic Languages, DLS 2007, pp. 53–64. ACM (2007)
4. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 221–239. Springer, Heidelberg (2006). [https://doi.org/10.1007/11823230\\_15](https://doi.org/10.1007/11823230_15)
5. Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static analysis and verification of aerospace software by abstract interpretation. In: AIAA Infotech@Aerospace, number 2010–3385 in AIAA, pp. 1–38. AIAA (American Institute of Aeronautics and Astronautics), April 2010
6. Bodin, M., Chargueraud, A., Filaretto, D., Gardner, P., Maffei, S., Naudziuniene, D., Schmitt, A., Smith, G.: A trusted mechanised JavaScript specification. SIGPLAN Not. **49**(1), 87–100 (2014)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL 1977), pp. 238–252. ACM, January 1977
8. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Conference Record of the 5th Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL 1978), pp. 84–97. ACM (1978)
9. Standard ECMA-262. ECMAScript 2017 Language Specification, 8th edn, June 2017
10. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of JavaScript. In: D’Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 126–150. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-14107-2\\_7](https://doi.org/10.1007/978-3-642-14107-2_7)
11. Guth, D.: A formal semantics of Python 3.3. Master’s thesis, University of Illinois at Urbana-Champaign, July 2013
12. Hassan, M.: SMT-based static type inference for Python 3. Bachelor thesis, ETH Zürich, Department of Computer Science (2017)
13. Jensen, S.H., Jonsson, P.A., Möller, A.: Remediating the eval that men do. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, pp. 34–44. ACM (2012)
14. Jensen, S.H., Möller, A., Thiemann, P.: Type analysis for JavaScript. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 238–255. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-03237-0\\_17](https://doi.org/10.1007/978-3-642-03237-0_17)
15. Kashyap, V., Dewey, K., Kuefner, E.A., Wagner, J., Gibbons, K., Sarracino, J., Wiedermann, B., Hardekopf, B.: JSAI: a static analysis platform for JavaScript. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pp. 121–132. ACM (2014)
16. Madsen, M., Andreasen, E.: String analysis for dynamic field access. In: Cohen, A. (ed.) CC 2014. LNCS, vol. 8409, pp. 197–217. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-642-54807-9\\_12](https://doi.org/10.1007/978-3-642-54807-9_12)

17. Miné, A.: The octagon abstract domain. *Higher Order Symbol. Comput.* **19**(1), 31–100 (2006)
18. Oh, H., Heo, K., Lee, W., Lee, W., Yi, K.: Design and implementation of sparse global analyses for C-like languages. *SIGPLAN Not.* **47**(6), 229–238 (2012)
19. Poli, F.: A small step abstract interpreter for (desugared) Python. Master’s thesis, Università degli Studi di Padova, Dipartimento di Matematica (2016)
20. Politz, J.G., Martinez, A., Milano, M., Warren, S., Patterson, D., Li, J., Chitipothu, A., Krishnamurthi, S.: Python: the full monty. *SIGPLAN Not.* **48**(10), 217–232 (2013)
21. Python Software Foundation. The Python language reference, 3.6 edn (2017). <https://docs.python.org/3.6/reference>
22. Ranson, J.F., Hamilton, H.J., Fong, P.W.L.: A semantics of Python in Isabelle/HOL. Technical report, Department of Computer Science, University of Regina, December 2008
23. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: *Program Flow Analysis: Theory and Applications*, pp. 189–234. Prentice-Hall, Upper Saddle River (1981)
24. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: understanding object-sensitivity. *SIGPLAN Not.* **46**(1), 17–30 (2011)
25. Smeding, G.J.: An executable operational semantics for Python. Master’s thesis, Universiteit Utrecht (2009)
26. Spoto, F.: Julia: a generic static analyser for the Java bytecode. In: *Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP 2005)*, p. 17, July 2005



# Model-Based Testing for General Stochastic Time

Marcus Gerhold<sup>(✉)</sup>, Arnd Hartmanns, and Mariëlle Stoelinga

University of Twente, Enschede, The Netherlands  
{m.gerhold,a.hartmanns}@utwente.nl, marielle@cs.utwente.nl

**Abstract.** Many systems are inherently stochastic: they interact with unpredictable environments or use randomised algorithms. Then classical model-based testing is insufficient: it only covers functional correctness. In this paper, we present a new model-based testing framework that additionally covers the stochastic aspects in hard and soft real-time systems. Using the theory of stochastic automata for specifications, test cases and a formal notion of conformance, it provides clean mechanisms to represent underspecification, randomisation, and stochastic timing. Supporting arbitrary continuous and discrete probability distributions, the framework generalises previous work based on purely Markovian models. We cleanly define its theoretical foundations, and then outline a practical algorithm for statistical conformance testing based on the Kolmogorov-Smirnov test. We exemplify the framework's capabilities and tradeoffs by testing timing aspects of the Bluetooth device discovery protocol.

## 1 Introduction

Model-based testing (MBT) [29] is a technique to automatically generate, execute and evaluate test suites on black-box *implementations under test* (IUT). The theoretical ingredients of an MBT framework are a formal *model* that specifies the desired system behaviour, usually in terms of (some extension of) input-output transition systems; a notion of *conformance* that specifies when an IUT is considered a valid implementation of the model; and a precise definition of what a *test case* is. For the framework to be applicable in practice, we also need algorithms to *derive* test cases from the model, *execute* them on the IUT, and *evaluate* the results, i.e. decide conformance. They need to be sound (i.e. every implementation that fails a test case does not conform to the model), and ideally also complete (i.e. for every non-conforming implementation, there theoretically exists a failing test case). MBT is attractive due to its high degree of automation: given a model, the otherwise labour-intensive and error-prone derivation, execution and evaluation steps can be performed in a fully automatic way.

Model-based testing originally gained prominence for input-output transition systems (IOTS) using the *ioco* relation for *input-output conformance* [28]. IOTS partition the observable actions of the IUT (and thus of the model and test cases)

---

This work is supported by projects 3TU.BSR, NWO BEAT and NWO SUMBAT.

into *inputs* (or *stimuli*) that can be provided at any time, e.g. pressing a button or receiving a network message, and *outputs* that are signals or activities that the environment can observe, e.g. delivering a product or sending a network message. IOTS models may include nondeterministic choices, allowing underspecification: the IUT may implement any or all of the modelled alternatives. MBT with IOTS tests for *functional* correctness: the IUT only exhibits behaviours allowed by the model. In the presence of nondeterminism, the IUT is allowed to use *any* deterministic or randomised policy to decide between the specified alternatives.

Stochastic behaviour and requirements are an important aspect of today's complex systems: network protocols extensively rely on randomised algorithms, cloud providers commit to service level agreements, probabilistic robotics [26] allows the automation of complex tasks via simple randomised strategies (as seen in e.g. vacuuming and lawn mowing robots), and we see a proliferation of probabilistic programming languages [15]. Stochastic systems must satisfy stochastic requirements. Consider the example of exponential backoff in Ethernet: an adapter that, after a collision, sometimes retransmits earlier than prescribed by the standard may not impact the overall functioning of the network, but may well gain an unfair advantage in throughput at the expense of overall network performance. In the case of cloud providers, the service level agreements are inherently stochastic when guaranteeing a certain availability (i.e. *average* uptime) or a certain distribution of maximum response times for different tasks. This has given rise to extensive research in stochastic *model checking* techniques [18]. However, in practice, *testing* remains the dominant technique to evaluate and certify systems outside of a limited area of highly safety-critical applications.

In this paper, we present a new MBT framework based on input-output stochastic automata (IOSA) [9], which are transition systems augmented with discrete probabilistic choices and timers whose expiration is governed by general probability distributions. By using IOSA models, we can quantitatively specify stochastic aspects of a system, in particular w.r.t. timing. We support discrete as well as continuous probability distributions, so our framework is suitable for both hard and soft real-time requirements. Since IOSA extend transition systems, nondeterminism is available for underspecification as usual. Test cases are IOSA, too, so they can naturally include waiting. We formally define the notions of stochastic ioco (*sa-ioco*), and of test cases as a restriction of IOSA (Sect. 3). We then outline practical algorithms for test generation and *sa-ioco* conformance testing (Sect. 4). The latter combines per-trace functional verdicts as in standard ioco with a statistical evaluation that builds upon the Kolmogorov-Smirnov test [17]. While our theory of IOSA and *sa-ioco* is very general w.r.t. supported probability distributions and nondeterminism, we need to assume some restrictions to arrive at practically feasible algorithms. We finally exemplify our framework's capabilities and its inherent tradeoffs by testing timing aspects of different implementation variants of the Bluetooth device discovery protocol (Sect. 5).

*Related Work.* Our new *sa-ioco* framework generalises two previous stochastic MBT approaches: the *pioco* framework [13] for probabilistic automata (or Markov decision processes) and *marioco* [14] for Markov automata (MA [12],



which extend continuous-time Markov chains with nondeterminism). The former only supports discrete probabilistic choices and has no notion of time at all. The latter operates under the assumption that all timing is memoryless, i.e. all delays are exponentially distributed and fully characterised by means.

Early influential work had only deterministic time [3, 19, 21], later extended with timeouts/quiescence [4]. Probabilistic testing preorders and equivalences are well-studied [7, 10, 24]. Probabilistic bisimulation via hypothesis testing was first introduced in [20]. Our work is largely influenced by [5], which introduced a way to compare trace frequencies with collected samples. Closely related is work on stochastic finite state machines [16, 23]: stochastic delays are specified similarly, but discrete probability distributions over target states are not included.

## 2 Background

*Notation.*  $\mathbb{R}^+$  and  $\mathbb{R}_0^+$  are the positive and non-negative real numbers. For a given set  $\Omega$ , its powerset is  $\mathcal{P}(\Omega)$ . A multiset is written as  $\{\! \{ \dots \} \}$ .  $\text{Dist}(\Omega)$  is the set of *probability distributions* over  $\Omega$ : functions  $\mu \in \Omega \rightarrow [0, 1]$  s.t.  $\text{support}(\mu) \stackrel{\text{def}}{=} \{\omega \in \Omega \mid \mu(\omega) > 0\}$  is countable and  $\sum_{\omega \in \text{support}(\mu)} \mu(\omega) = 1$ .  $\Omega$  is *measurable* if it is endowed with a  $\sigma$ -algebra  $\sigma(\Omega)$ : a collection of *measurable* subsets of  $\Omega$ . A *probability measure* over  $\Omega$  is a function  $\mu \in \sigma(\Omega) \rightarrow [0, 1]$  s.t.  $\mu(\Omega) = 1$  and  $\mu(\cup_{i \in I} B_i) = \sum_{i \in I} \mu(B_i)$  for any countable index set  $I$  and pairwise disjoint measurable sets  $B_i \subseteq \Omega$ .  $\text{Prob}(\Omega)$  is the set of probability measures over  $\Omega$ . Each  $\mu \in \text{Dist}(\Omega)$  induces a probability measure. Let  $\text{Val} \stackrel{\text{def}}{=} V \rightarrow \mathbb{R}_0^+$  be the set of valuations for an (implicit) set  $V$  of (non-negative real-valued) variables.  $\mathbf{0} \in \text{Val}$  assigns value zero to all variables. For  $X \subseteq V$  and  $v \in \text{Val}$ , we write  $v[X]$  for the valuation defined by  $v[X](x) = 0$  if  $x \in X$  and  $v[X](y) = v(y)$  otherwise. For  $t \in \mathbb{R}_0^+$ ,  $v + t$  is the defined by  $(v + t)(x) = v(x) + t$  for all  $x \in V$ .

**Stochastic automata** extend Markov decision processes with stochastic *clocks*: real-valued variables that increase synchronously with rate 1 over time and expire some random amount of time after they have been *restarted*. We define SA with input/output actions along the lines of [9]:

**Definition 1.** An input-output stochastic automaton (IOSA) is a 6-tuple  $\mathcal{I} = \langle \text{Loc}, \mathcal{C}, \mathcal{A}, E, F, \ell_{\text{init}} \rangle$  where  $\text{Loc}$  is a countable set of locations,  $\mathcal{C}$  is a finite set of clocks,  $\mathcal{A} = \mathcal{A}^I \uplus \mathcal{A}^O$  is the finite action alphabet partitioned into inputs in  $\mathcal{A}^I$  (marked by a ? suffix) and outputs in  $\mathcal{A}^O$  (marked by a ! suffix),  $E \in \text{Loc} \rightarrow \mathcal{P}(\text{Edges})$  with  $\text{Edges} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{C}) \times \mathcal{A} \uplus \{\tau, \delta\} \times \text{Dist}(T)$  and  $T \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{C}) \times \text{Loc}$  is the edge function mapping each location to a finite set of edges that in turn consist of a guard set, a label that may be the internal action  $\tau$  or quiescence  $\delta$ , and a distribution over targets in  $T$  consisting of a restart set of clocks and target locations,  $F \in \mathcal{C} \rightarrow \text{Prob}(\mathbb{R}_0^+)$  is the delay measure function that maps each clock to a probability measure, and  $\ell_{\text{init}} \in \text{Loc}$  is the initial location.  $\mathcal{I}$  is input-enabled if  $\forall \ell \in \text{Loc}, a \in \mathcal{A}^I \exists \mu: \langle \emptyset, a, \mu \rangle \in E(\ell)$ .  $\mathcal{I}$  is closed if  $\mathcal{A}^I = \emptyset$ .

We also write  $\ell \xrightarrow{G,a}_E \mu$  for  $\langle G, a, \mu \rangle \in E(\ell)$ . Whenever an IOSA  $\mathcal{I}_i$  or  $\mathcal{S}_i$  (where index  $i$  may be absent) is given in the remainder of this paper, it has the form  $\langle Loc_i, \mathcal{C}_i, \mathcal{A}_i, E_i, F_i, \ell_{init_i} \rangle$  unless noted otherwise. Intuitively, a stochastic automaton starts its execution in the initial location with all clocks expired. An edge  $\ell \xrightarrow{G,a}_E \mu$  may be taken only if all clocks in its guard set  $G$  are expired. If *any* output edge (i.e. with  $a \in \mathcal{A}^O$ ) is enabled, *some* edge must be taken (i.e. all outputs are *urgent*). When an edge is taken, (1) its action is  $a$ , (2) we select a target  $\langle R, \ell' \rangle \in T$  randomly according to  $\mu$ , (3) all clocks in  $R$  are restarted and other expired clocks remain expired, and (4) we move to successor location  $\ell'$ . There, another edge may be taken immediately or we may need to wait until some further clocks expire, and so on. When a clock  $c$  is restarted, the time until it expires is chosen randomly according to the probability measure  $F(c)$ .

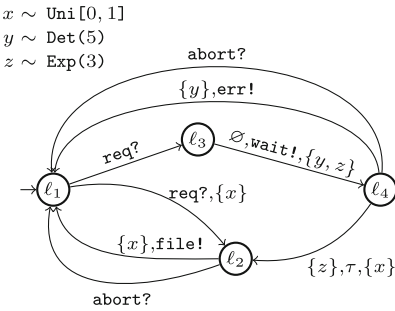


Fig. 1. File server specification.

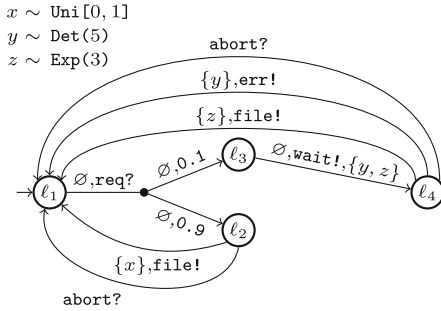


Fig. 2. File server implementation.

*Example 1.* Figure 1 shows an example IOSA specifying the behaviour of a file server with archival storage. We omit empty restart sets and the empty guard sets of inputs. Upon receiving a request in the initial location  $\ell_1$ , an implementation may either move to  $\ell_2$  or  $\ell_3$ . The latter represents the case of a file in archive: the server must immediately deliver a `wait!` notification and then attempt to retrieve the file from the archive. Clocks  $y$  and  $z$  are restarted, and used to specify that retrieving the file shall take on average  $\frac{1}{3}$  of a time unit, exponentially distributed, but no more than 5 time units. In location  $\ell_4$ , there is thus a race between retrieving the file and a deterministic timeout. In case of timeout, an error message (action `err!`) is returned; otherwise, the file can be delivered as usual from location  $\ell_2$ . Clock  $x$  is used to specify the transmission time of the file: it shall be uniformly distributed between 0 and 1 time units.

In Fig. 2, we show an implementation of this specification. 1 out of 10 files randomly requires to be fetched from the archive. This is allowed by the specification: it is one particular (randomised) resolution of the nondeterministic choice, i.e. underspecification, defined in  $\ell_1$ . The implementation also manages to transmit files from archive directly while fetching them, as evidenced by the direct edge from  $\ell_4$  back to  $\ell_1$  labelled `file!`. This violates the timing prescribed by the specification, and must be detected by an MBT procedure for IOSA.

**Definition 2.** Given IOSA  $\mathcal{I}_1, \mathcal{I}_2$  with  $\mathcal{C}_1 \cap \mathcal{C}_2 = \emptyset$ , and  $M \subseteq \mathcal{A}_1 \times \mathcal{A}_2$ , their parallel composition is  $\mathcal{I}_1 \parallel \mathcal{I}_2 \stackrel{\text{def}}{=} \langle \text{Loc}_1 \times \text{Loc}_2, \mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{A}_{\parallel}, E_{\parallel}, F_1 \cup F_2, \langle \ell_{\text{init}_1}, \ell_{\text{init}_2} \rangle \rangle$  where  $\mathcal{A}_{\parallel} \stackrel{\text{def}}{=} \mathcal{A}_{\parallel}^I \uplus \mathcal{A}_{\parallel}^O$  with outputs  $\mathcal{A}_{\parallel}^O = \mathcal{A}_1^O \cup \mathcal{A}_2^O$  and inputs  $\mathcal{A}_{\parallel}^I = (\mathcal{A}_1^I \cup \mathcal{A}_2^I) \setminus (\{a_I \in \mathcal{A}_1^I \mid \exists a_O \in \mathcal{A}_2^O : \langle a_I, a_O \rangle \in M\} \cup \{a_I \in \mathcal{A}_2^I \mid \exists a_O \in \mathcal{A}_1^O : \langle a_O, a_I \rangle \in M\})$  and  $E_{\parallel}$  is the smallest edge function satisfying the inference rules

$$\frac{\ell_1 \xrightarrow{G, a} E_1 \mu \quad a = \tau \vee \nexists a_2 \in \mathcal{A}_2 : \langle a, a_2 \rangle \in M}{\langle \ell_1, \ell_2 \rangle \xrightarrow{G, a} E_{\parallel} \{ \langle R, \langle \ell'_1, \ell_2 \rangle \rangle \mapsto \mu(\langle R, \ell'_1 \rangle) \mid R \subseteq \mathcal{C}, \ell'_1 \in \text{Loc}_1 \}} \quad (\text{indep}_1)$$

$$\frac{\ell_1 \xrightarrow{G_1, a_1} E_1 \mu_1 \quad \ell_2 \xrightarrow{G_2, a_2} E_2 \mu_2 \quad a_1 \in \mathcal{A}_1^O \wedge \langle a_1, a_2 \rangle \in M}{\langle \ell_1, \ell_2 \rangle \xrightarrow{G_1 \cup G_2, a_1} E_{\parallel} \{ \langle R_1 \cup R_2, \langle \ell'_1, \ell'_2 \rangle \rangle \mapsto \mu(\langle R_1, \ell'_1 \rangle) \cdot \mu(\langle R_2, \ell'_2 \rangle) \}} \quad (\text{sync}_1)$$

plus symmetric rules  $\text{indep}_2$  and  $\text{sync}_2$  for the corresponding steps of  $\mathcal{I}_2$ .

We use the convention that two actions  $a_1$  and  $a_2$  match, i.e.  $\langle a_1, a_2 \rangle \in M$ , if they are the same except for the suffix (e.g.  $\mathbf{a}!$  matches  $\mathbf{a}?$  but not  $\mathbf{b}?$  or  $\mathbf{a}!$ ).

**Definition 3.** The states of IOSA  $\mathcal{I}$  are  $S \stackrel{\text{def}}{=} \text{Loc} \times \text{Val} \times \text{Val}$ . Each  $\langle \ell, v, x \rangle \in S$  consists of the current location  $\ell$  and the values  $v$  and expiration times  $x$  of all clocks. The set of paths of  $\mathcal{I}$  is  $\text{Paths}_{\mathcal{I}} \stackrel{\text{def}}{=} S \times (\mathbb{R}_0^+ \times \text{Edges} \times \mathcal{P}(\mathcal{C}) \times S)^\omega$  where the first state is  $\langle \ell_{\text{init}}, \mathbf{0}, \mathbf{0} \rangle$ .  $\text{Paths}_{\mathcal{I}}^{\text{fin}}$  is the set of all finite paths. For  $\pi \in \text{Paths}_{\mathcal{I}}^{\text{fin}}$ ,  $\text{last}(\pi)$  is its last state, and its length is the number of edges with actions  $\neq \tau$ .

**Definition 4.** A scheduler of a closed IOSA  $\mathcal{I}$  is a measurable function  $\mathfrak{S} \in \text{Sched}(\mathcal{I}) \stackrel{\text{def}}{=} \text{Paths}_{\mathcal{I}}^{\text{fin}} \rightarrow \text{Dist}(\text{Edges} \cup \{\perp\})$  such that  $\mathfrak{S}(\pi)(\langle G, a, \mu \rangle) > 0$  with  $\text{last}(\pi) = \langle \ell, v, x \rangle$  implies  $\ell \xrightarrow{G, a} \mu$  and  $\text{Ex}(G, v + t, x)$  where  $t \in \mathbb{R}_0^+$  is the minimal delay for which  $\nexists t' \in [0, t[ : \bigvee_{\ell' \xrightarrow{G', a'} \mu'} \text{Ex}(G, v + t', x)$ . We define  $\text{Ex}(G, v, x) \stackrel{\text{def}}{=} \forall c \in G : v(c) \geq x(c)$ , i.e. all clocks in  $G$  are expired.  $\mathfrak{S}(\pi)(\perp)$  is the probability to halt.  $\mathfrak{S}$  is of length  $k \in \mathbb{N}$  if  $\mathfrak{S}(\pi)(\perp) = 1$  for all paths  $\pi$  of length  $\geq k$ .  $\text{Sched}(\mathcal{I}, k)$  is the set of all schedulers of  $\mathcal{I}$  of length  $k$ .

A scheduler can only choose between the edges enabled at the points where *any* edge just became enabled in a closed IOSA. It removes all nondeterminism. The probability of each step on a path is then given by the *step probability function*:

**Definition 5.** Given IOSA  $\mathcal{I}$  and  $\mathfrak{S} \in \text{Sched}(\mathcal{I})$ , the step probability function

$$\begin{aligned} & Pr_{\mathfrak{S}} \in \text{Paths}_{\mathcal{I}}^{\text{fin}} \rightarrow \text{Prob}(\{\perp\} \cup (\mathbb{R}_0^+ \times \text{Edges} \times \mathcal{P}(\mathcal{C}) \times S)) \\ & \text{is defined by } Pr_{\mathfrak{S}}(\pi)(\perp) = \mathfrak{S}(\pi)(\perp) \text{ and, for } \pi \text{ with } \text{last}(\pi) = \langle \ell, v, x \rangle, \\ & Pr_{\mathfrak{S}}(\pi)([t_1, t_2] \times E_{Pr} \times C_{Pr} \times S_{Pr}) = \\ & \mathbb{1}_{t \in [t_1, t_2]} \cdot \sum_{e = \langle G, a, \mu \rangle \in E_{Pr}} \mathfrak{S}(\pi)(e) \cdot \sum_{C \in C_{Pr}, \ell' \in \text{Loc}} \mu(\langle C, \ell' \rangle) \cdot \int_{\langle \ell', v', x' \rangle \in S_{Pr}} X_C^x(v', x') \end{aligned}$$

where  $t$  is the minimal delay in  $\ell$  as in Definition 4 and

$$X_C^x(v', x') = \mathbb{1}_{v'=(v+t)[C]} \prod_{c \in C} \begin{cases} 1 & \text{if } c \notin C \wedge x(c) = x'(c) \\ 0 & \text{if } c \notin C \wedge x(c) \neq x'(c) \\ F(c)(t_2) - F(c)(t_1) & \text{if } c \in C. \end{cases}$$

The step probability function induces a probability measure over  $\text{Paths}_{\mathcal{I}}$ . As is usual, we restrict to schedulers that let time diverge with probability 1.

A path lets us follow exactly how an IOSA was traversed. Traces represent the knowledge of external observers. In particular, they cannot see the values of individual clocks, but only the time passed since the run started. Formally:

**Definition 6.** *The trace of a (finite) path  $\pi$  is its projection  $tr(\pi)$  to the delays in  $\mathbb{R}_0^+$  and the actions in  $\mathcal{A}$ .  $\tau$ -steps are omitted and their delays are added to that of the next visible action. The set of traces of  $\mathcal{I}$  is  $Traces_{\mathcal{I}}$ . An abstract trace in  $AbsTraces_{\mathcal{I}}$  is a sequence  $\Sigma = I_1 a_1 I_2 a_2 \dots$  with the  $I_i$  closed intervals over  $\mathbb{R}_0^+$ . Finite (abstract) traces are defined analogously.  $Traces_{\mathcal{I}}^{\max}$  is the set of maximal finite traces for  $\mathcal{I}$  with terminal locations.  $\sigma$  represents the set of traces  $\{t_1 a_1 \dots \mid t_i \in I_i\}$ . We identify trace  $t_1 a_1 \dots$  with abstract trace  $[0, t_1] a_1 \dots$ .*

We can define the *trace distribution* for an IOSA  $\mathcal{I}$  and a scheduler as the probability measure over traces (using abstract traces to construct the corresponding  $\sigma$ -algebra) induced by the probability measure over paths in the usual way. The set of all finite trace distributions is  $Trd(\mathcal{I})$ . It induces an equivalence relation  $\equiv_{TD}$ : two IOSA  $\mathcal{I}$  and  $\mathcal{S}$  are trace distribution equivalent, written  $\mathcal{I} \equiv_{TD} \mathcal{S}$ , if and only if  $Trd(\mathcal{I}) = Trd(\mathcal{S})$ . A trace distribution is of length  $k \in \mathbb{N}$  if it is based on a scheduler of length  $k$ . The set of all such trace distributions is  $Trd(\mathcal{I}, k)$ .

### 3 Stochastic Testing Theory

We define the theoretical concepts of our MBT framework: test cases, the *sa-ioco* conformance relation, the evaluation of test executions, and correctness notions. The specifications  $\mathcal{S}$  are IOSA as in Definition 1, and we equally assume the IUT to be an input-enabled IOSA  $\mathcal{I}$  with the same alphabet as  $\mathcal{S}$ .

#### 3.1 Test Cases

A test case describes the possible behaviour of a tester. The advantage of MBT over manual testing is that test cases can be automatically generated from the specification, and automatically executed on an implementation. In each step of the execution, the tester may either (1) send an input to the IUT, (2) wait to observe output, or (3) stop testing. A single test may provide multiple options, giving rise to multiple concrete testing sequences. It may also prescribe different reactions to different outputs. Formally, test cases for a specification  $\mathcal{S}$  are IOSA whose inputs are the outputs of  $\mathcal{S}$  and vice-versa. The parallel composition of either  $\mathcal{S}$  or  $\mathcal{I}$  with a test case thus results in a closed IOSA. By including discrete probability distributions on edges, IOSA allow making the probabilities of the three choices (input, wait, stop) explicit<sup>1</sup>. Moreover, we can use clocks for explicit waiting times in test cases. Sending input can hence be delayed, which is especially beneficial to test race conditions. A test can also react to *no* output being supplied, modelled by *quiescence*  $\delta$ , and check if that was expected.

<sup>1</sup> Tests are often *implicitly* generated probabilistically in classic *ioco* settings, too, without the support to make this explicit in the underlying theory. We fill this gap.

**Definition 7.** A test  $\mathcal{T}$  for a specification  $\mathcal{S}$  with alphabet  $\mathcal{A}^I \uplus \mathcal{A}^O$  is an IOSA  $\langle \text{Loc}, \mathcal{C}, \mathcal{A}^O \uplus \mathcal{A}^I, E, F, \ell_{\text{init}} \rangle$  that has the specification's outputs as inputs and vice-versa, and that is a finite, internally deterministic, acyclic and connected tree such that for every location  $\ell \in \text{Loc}$ , we either have  $E(\ell) = \emptyset$  (stop testing), or  $\forall \ell \xrightarrow{G, a} \mu: a = \tau$  (an internal decision), or if  $\exists \ell \xrightarrow{G, a} \mu: a \in \mathcal{A}^I \cup \{\delta\}$  (we can send an input or observe quiescence) then:  $\forall a_O \in \mathcal{A}^O: \exists G', \mu': \ell \xrightarrow{G', a_O} \mu'$  (all outputs can be received) and  $\forall \ell \xrightarrow{G', a'} \mu': a' \in \mathcal{A}^O \vee a' = a$  (we cannot send a different input or observe quiescence in addition to an input). Whenever  $\mathcal{T}$  sends an input, this input must be present in  $\mathcal{S}$ , too, i.e.

$$\forall \sigma \in \text{Traces}_{\mathcal{T}}^{\text{fin}} \text{ with } \sigma = \sigma_1 t a \sigma_2 \text{ and } a \in \mathcal{A}^I: \sigma_1 t a \in \text{Traces}_{\mathcal{S}}^{\text{fin}}.$$

### 3.2 Stochastic Input-Output Conformance and Annotations

Trace distribution equivalence  $\equiv_{TD}$  is the probabilistic counterpart of *trace equivalence* for transition systems: it shows that there is a way for the traces of two different models, e.g. the IOSA  $\mathcal{S}$  and  $\mathcal{I}$ , to all have the same probability via *some* resolution of nondeterminism. However, trace equivalence or inclusion is too fine as a conformance relation for testing [27]. The *ioco* relation [28] for *functional* conformance solves the problem of fineness by allowing underspecification of functional behaviour: an implementation  $\mathcal{I}$  is *ioco*-conforming to a specification  $\mathcal{S}$  if every experiment derived from  $\mathcal{S}$  executed on  $\mathcal{I}$  leads to an output that was foreseen in  $\mathcal{S}$ . Formally:

$$\mathcal{I} \sqsubseteq_{ioco} \mathcal{S} \Leftrightarrow \forall \sigma \in \text{Traces}_{\mathcal{S}}^{\text{fin}}: \text{out}_{\mathcal{I}}(\sigma) \subseteq \text{out}_{\mathcal{S}}(\sigma)$$

where  $\text{out}_{\mathcal{I}}(\sigma)$  is the set of outputs in  $\mathcal{I}$  that is enabled after the trace  $\sigma$ .

*Stochastic ioco.* To extend *ioco* testing to IOSA, we need an auxiliary concept that mirrors trace prefixes stochastically: Given a trace distribution  $D$  of length  $k$ , and a trace distribution  $D'$  of length greater or equal than  $k$ , we say  $D$  is a *prefix* of  $D'$ , written  $D \sqsubseteq_k D'$ , if both assign the same probability to all abstract traces of length  $k$ . We can then define:

**Definition 8.** Let  $\mathcal{S}$  and  $\mathcal{I}$  be two IOSA. We say  $\mathcal{I}$  is *sa-ioco-conforming* to  $\mathcal{S}$ , written  $\mathcal{I} \sqsubseteq_{ioco}^{sa} \mathcal{S}$ , if and only if for all tests  $\mathcal{T}$  for  $\mathcal{S}$  we have  $\mathcal{I} \parallel \mathcal{T} \sqsubseteq_{TD} \mathcal{S} \parallel \mathcal{T}$ .

Intuitively,  $\mathcal{I}$  is conforming if, no matter how it resolves nondeterminism (i.e. underspecification) under a concrete test,  $\mathcal{S}$  can mimic its behaviour by resolving nondeterminism for the same test, such that all traces have the same probability.

The original *ioco* relation takes an experiment derived from the specification and executes it on the implementation. While Definition 8 does not directly mirror this, the property implicitly carries over from the parallel composition with tests specifically *designed for* specifications: an input is provided to the IUT if that input is present in the specification model only.

*Open schedulers.* The above difference in approach between *ioco* and *sa-ioco* is due to schedulers and their resulting trace distributions being solely defined for closed systems in our work (cf. Definition 4). An alternative is to also define

them for open systems. However, where schedulers for closed systems choose discretely between possible actions, their counterparts for open systems additionally schedule over continuous time, i.e. *when* an action is to be taken. This poses an additional layer of difficulty in tracing which scheduler was likely used to resolve nondeterminism, which we need to do in our *a posteriori* statistical analysis of the testing results (see Sect. 3.3).

Moreover, it is known [1, 6, 25] that trace distributions of probabilistic systems under “open” schedulers are not compositional in general, i.e.  $A \sqsubseteq_{TD} B$  does not imply  $A \parallel C \sqsubseteq_{TD} B \parallel C$ . This would mean that, even when an implementation conforms to a specification, the execution of a probabilistic test case might tamper with the observable probabilities and yield an untrustworthy verdict. A general counterexample for the above implication is presented in [25], where however there is no requirement on input-enabledness of the composable systems. Our framework requires both implementation and test case to be input-enabled, cf. Definitions 7 and 8. The authors of [1] provide a counterexample for synchronous systems even in the presence of input-enabledness. Our framework works with input-enabled *asynchronous* systems; we thus believe that *sa-ioco* could also be defined in a way that more closely resembles the original definition of *ioco* by using open schedulers, but care has to be taken in defining those schedulers in the right way. We thus designed *sa-ioco* conservatively such that it is only based on trace semantics of closed systems, while still maintaining the possibility of underspecification as in *ioco* due to the way tests are used.

*Annotations.* To assess whether observed behaviour is functionally correct, each complete trace of a test is annotated with a verdict: all leaf locations of test cases are labelled with either *pass* or *fail*. We annotate exactly the traces that are present in the specification with the *pass* verdict; formally:

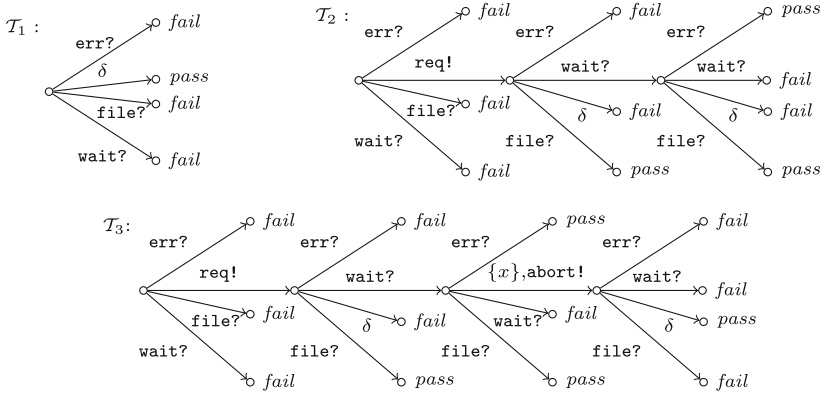
**Definition 9.** *Given a test  $\mathcal{T}$  for specification  $\mathcal{S}$ , its test annotation is the function  $ann \in \text{Traces}_{\mathcal{T}}^{\max} \rightarrow \{\text{pass}, \text{fail}\}$  such that  $ann(\sigma) = \text{fail}$  if and only if  $\exists \varrho \in \text{Traces}_{\mathcal{S}}^{\text{fin}}, t \in \mathbb{R}_0^+, a \in \mathcal{A}^O: \varrho t a$  is a prefix of  $\sigma \wedge \varrho t a \notin \text{Traces}_{\mathcal{S}}^{\text{fin}}$ .*

Annotations decide functional correctness only. The correctness of discrete probability choices and stochastic clocks is assessed in a separate second step.

*Example 2.* Figure 3 presents three test cases for the file server specification of Ex. 1.  $\mathcal{T}_1$  uses the quiescence observation  $\delta$  to assure no output is given in the initial state.  $\mathcal{T}_2$  tests for *eventual* delivery of the file, which may be in archive, requiring the intermediate *wait!* notification, or may be sent directly.  $\mathcal{T}_3$  utilises a clock on the *abort!* transition: it waits for some time (depending on what  $\mathcal{T}_3$  specifies for  $F(x)$ ) before sending the input. This highlights the ability to test for race conditions, or for the possibility of a file arrival before a specified time.

### 3.3 Test Execution and Sampling

We test stochastic systems: executing a test case  $\mathcal{T}$  once is insufficient to establish *sa-ioco* conformance. We need many executions for an overall statistical verdict



**Fig. 3.** Three test cases  $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$  for the file server specification.

about the stochastic behaviour in addition to the functional verdict obtained from the annotation on each execution. As establishing the functional verdict is the same as in standard ioco testing, we focus on the statistical evaluation here.

*Sampling.* We perform a statistical hypothesis test on the implementation based on the outcome of a push-button experiment in the sense of [22]. Before the experiment, we fix the parameters for sample length  $k \in \mathbb{N}$  (the length of the individual test executions), sample width  $m \in \mathbb{N}$  (how many test executions to observe), and level of significance  $\alpha \in (0, 1)$ . The latter is a limit for the *statistical error of first kind*, i.e. the probability of rejecting a correct implementation. The statistical analysis is performed after collecting the sample for the chosen parameters, while functional correctness is checked during the sampling process.

*Frequencies.* Our goal is to determine the deviation of a sample of traces  $O = \{\sigma_1, \dots, \sigma_m\}$  taken from  $\mathcal{I} \parallel \mathcal{T}$  vs. the results expected for  $\mathcal{S} \parallel \mathcal{T}$ . If it is too large,  $O$  was likely not generated by an IOSA conforming to  $\mathcal{S}$  and we reject  $\mathcal{I}$ . If the deviation is within bounds depending on  $k, m$  and  $\alpha$ , we have no evidence to suspect an underlying IOSA other than  $\mathcal{S}$  and accept  $\mathcal{I}$  as a conforming IUT. We compare the frequencies of traces in  $O$  with their probabilities according to  $\mathcal{S} \parallel \mathcal{T}$ . Since  $\mathcal{I}$  is a concrete implementation, the scheduler is the same for all executions, resulting in trace distribution  $D$  for  $\mathcal{I} \parallel \mathcal{T}$  and the probability of abstract trace  $\Sigma$  is given directly by  $D(\Sigma)$ . We define  $freq_O(\Sigma) \stackrel{\text{def}}{=} |\{\sigma \in O \mid \sigma \in \Sigma\}|/m$ , i.e. the fraction of traces in  $O$  that are in  $\Sigma$ .  $\mathcal{I}$  is rejected on statistical evidence if the distance of the two measures  $D$  and  $freq_O$  exceeds a threshold based on  $\alpha$ .

*Acceptable outcomes.* We accept a sample  $O$  if  $freq_O$  lies within some radius, say  $r_\alpha$ , around  $D$ . To minimise the error of false acceptance, we choose the smallest  $r_\alpha$  that guarantees that the error of false rejection is not greater than  $\alpha$ , i.e.

$$r_\alpha \stackrel{\text{def}}{=} \inf \{ r \in \mathbb{R}^+ \mid D(freq^{-1}(B_r(D))) > 1 - \alpha \}, \tag{1}$$

where  $B_y(x)$  is the closed ball centred at  $x \in X$  with radius  $y \in \mathbb{R}^+$  and  $X$  a metric space. The set of all measures defines a metric space together with the total variation distance of measures  $dist(u, v) \stackrel{\text{def}}{=} \sup_{\sigma \in (\mathbb{R}_0^+ \times \mathcal{A})^k} |u(\sigma) - v(\sigma)|$ .

**Definition 10.** For  $k, m \in \mathbb{N}$  and  $\mathcal{I} \parallel \mathcal{T}$ , the observations under a trace distribution  $D \in \text{Trd}(\mathcal{I} \parallel \mathcal{T}, k)$  of level of significance  $\alpha \in (0, 1)$  are given by the set

$$\text{Obs}(D, \alpha, k, m) = \{ O \in (\mathbb{R}_0^+ \times \mathcal{A})^{k \times m} \mid dist(\text{freq}_O, D) \leq r_\alpha \}.$$

The set of observations of  $\mathcal{I} \parallel \mathcal{T}$  with  $\alpha \in (0, 1)$  is then given by the union over all trace distributions of length  $k$ , and is denoted  $\text{Obs}(\mathcal{I} \parallel \mathcal{T}, \alpha, k, m)$ .

These sets limit the statistical error of first and second kind as follows: if a sample was generated under a trace distribution of  $\mathcal{I} \parallel \mathcal{T}$  or a trace distribution equivalent IOSA, we accept it with probability higher than  $1 - \alpha$ ; and for all samples generated under a trace distribution by non-equivalent IOSA, the chance of erroneously accepting it is smaller than some  $\beta_m$ , where  $\beta_m$  is unknown but minimal by construction, cf. (1). Note that  $\beta_m \rightarrow 0$  as  $m \rightarrow \infty$ , i.e. the error of accepting an erroneous sample decreases as sample size increases.

### 3.4 Test Evaluation and Correctness

**Definition 11.** Given an IOSA  $\mathcal{S}$ , an annotated test case  $\mathcal{T}$ ,  $k$  and  $m \in \mathbb{N}$ , and a level of significance  $\alpha \in (0, 1)$ , we define (1) the functional verdict as given by  $v_{\text{func}} \in \text{IOSA}^2 \rightarrow \{ \text{pass}, \text{fail} \}$  where  $v_{\text{func}}(\mathcal{I}, \mathcal{T}) = \text{pass}$  if and only if  $\forall \sigma \in \text{Traces}_{\mathcal{I} \parallel \mathcal{T}}^{\text{max}} \cap \text{Traces}_{\mathcal{T}}^{\text{max}}$ :  $\text{ann}(\sigma) = \text{pass}$ , and (2) the statistical verdict as given by  $v_{\text{prob}} \in \text{IOSA}^2 \rightarrow \{ \text{pass}, \text{fail} \}$  where  $v_{\text{prob}}(\mathcal{I}, \mathcal{T}) = \text{pass}$  iff  $\exists D \in \text{Trd}(\mathcal{S} \parallel \mathcal{T}, k)$  s.t.

$$D(\text{Obs}(\mathcal{I} \parallel \mathcal{T}, \alpha, k, m) \cap \text{Traces}_{\mathcal{S} \parallel \mathcal{T}}) > 1 - \alpha.$$

$\mathcal{I}$  passes a test suite if  $v_{\text{prob}}(\mathcal{I}, \mathcal{T}) = v_{\text{func}}(\mathcal{I}, \mathcal{T}) = \text{pass}$  for all annotated test cases  $\mathcal{T}$  of the test suite.

The above definition connects the previous two subsections to implement a *correct* MBT procedure for the *sa-ioco* relation introduced in Sect. 3.2. Correctness comprises *soundness* and *completeness* (or *exhaustiveness*): the first means that every conforming implementation passes a test, whereas the latter implies that there is a test case to expose every erroneous (i.e. nonconforming) implementation. A test suite can only be considered correct with a guaranteed (high) probability  $1 - \alpha$  (as inherent in Definition 11).

**Definition 12.** Let  $\mathcal{S}$  be a specification IOSA. Then a test case  $\mathcal{T}$  is sound for  $\mathcal{S}$  with respect to *sa-ioco* for every  $\alpha \in (0, 1)$  iff for every input enabled IOSA  $\mathcal{I}$  we have that  $\mathcal{I} \sqsubseteq_{\text{ioco}}^{\text{sa}} \mathcal{S}$  implies  $v_{\text{func}}(\mathcal{I}, \mathcal{T}) = v_{\text{prob}}(\mathcal{I}, \mathcal{T}) = \text{pass}$ .

Completeness of a test suite is inherently a theoretical result. Infinite behaviour of the IUT, for instance caused by loops, hypothetically requires a test suite of infinite size. Moreover, there remains a possibility of accepting an erroneous implementation by chance, i.e. making an error of second kind. However, the latter is bounded from above and decreases with increasing sample size.



**Definition 13.** *Let  $\mathcal{S}$  be a specification IOSA. Then a test suite is called complete for  $\mathcal{S}$  with respect to sa-ioco for every  $\alpha \in (0, 1)$  iff for every input-enabled IOSA  $\mathcal{I}$  we have that  $\mathcal{I} \not\sqsubseteq_{ioco}^{sa} \mathcal{S}$  implies the existence of a test  $\mathcal{T}$  in the test suite such that  $v_{func}(\mathcal{I}, \mathcal{T}) = fail$  or  $v_{prob}(\mathcal{S}, \mathcal{T}) = fail$ .*

## 4 Implementing Stochastic Testing

The previous section laid the theoretical foundations of our new IOSA-based testing framework. Several aspects were specified very abstractly, for which we now provide practical procedures. There are already several ways to generate, annotate and execute test cases in batch or on-the-fly in the classic *ioco* setting [28], which can be transferred to our framework. The statistical analysis of gathered sample data in MBT, on the other hand, is largely unexplored since few frameworks include probabilities or even stochastic timing. Determining verdicts according to Definition 11 requires concrete procedures to implement the statistical tests described in Sect. 3.3 with level of significance  $\alpha$ . We now present practical methods to evaluate test cases in line with this theory. In particular, we need to find a scheduler for  $\mathcal{S}$  that makes the observed traces  $O$  most likely, and test that the stochastic timing requirements are implemented correctly.

### 4.1 Goodness of Fit

Since our models neither comprise only *one specific distribution*, nor *one specific parameter* to test for, we resort to nonparametric goodness of fit tests. Nonparametric statistical procedures allow to test hypotheses that were designed for ordinal or nominal data [17], matching our intention of (1) testing the overall distribution of trace frequencies in a sample  $O = \{\sigma_1, \dots, \sigma_m\}$ , and (2) validating that the observed delays were drawn from the specified clocks and distributions. We use Pearson’s  $\chi^2$  test for (1) and multiple Kolmogorov-Smirnov tests for (2).

*Pearson’s  $\chi^2$  test* [17] compares empirical sample data to its expectations. It allows us to check the hypothesis that observed data indeed originates from a specified distribution. The cumulative sum of squared errors is compared to a critical value, and the hypothesis is rejected if the empiric value exceeds the threshold. We can thus check whether trace frequencies correspond to a specification under a certain trace distribution. For a finite trace  $\sigma = t_1 a_1 t_2 a_2 \dots t_k a_k$ , we define its timed closure as  $\bar{\sigma} \stackrel{\text{def}}{=} \mathbb{R}^+ a_1 \dots \mathbb{R}^+ a_k$ . Applying Pearson’s  $\chi^2$  is done in general via  $\chi^2 = \sum_{i=1}^n |\text{obs}_i - \text{exp}_i|^2 / \text{exp}_i$ , i.e. in our case

$$\chi^2 \stackrel{\text{def}}{=} \sum_{\bar{\sigma} \in \{\bar{\sigma} | \sigma \in O\}} \frac{(|\{\bar{\varrho} | \varrho \in O \wedge \bar{\varrho} = \bar{\sigma}\}| / m - D(\bar{\sigma}))^2}{D(\bar{\sigma})}. \tag{2}$$

We need to find a  $D$  that gives a high likelihood to a sample, i.e. such that  $\chi^2 < \chi_{crit}^2$ , where  $\chi_{crit}^2$  depends on  $\alpha$  and the degrees of freedom. The latter is given by the number of different timed closures in  $O$  minus 1. The critical values can be calculated or found in standard tables.

Recall that a trace distribution is based on a scheduler that resolves non-deterministic choices randomly. This turns (2) into a satisfaction problem of a

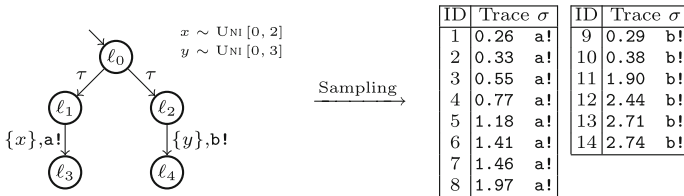
probability vector  $p$  over a rational function  $f(p)/g(p)$ , where  $f$  and  $g$  are polynomials. Finding a resolution such that  $\chi^2 < \chi_{crit}^2$  ensures that the error of rejecting a correct IUT is at most  $\alpha$ . This can be done via SMT solving.

*The Kolmogorov-Smirnov test.* While Pearson’s  $\chi^2$  test assesses the existence of a scheduler that explains the observed trace frequencies, it does not take into account the observed delays. For this purpose, we use the non-parametric Kolmogorov-Smirnov test [17] (the KS test). It assesses whether observed data matches a hypothesised *continuous* probability measure. We thus restrict the practical application of our approach to IOSA where the  $F(c)$  for all clocks  $c$  are continuous distributions. Let  $t_1, \dots, t_n$  be the delays observed for a certain edge over multiple traces in ascending order and  $F_n$  be the resulting step function, i.e. the right-continuous function  $F_n$  defined by  $F_n(t) = 0$  for  $t < t_1$ ,  $F_n(t) = n_i/n$  for  $t_i \leq t < t_{i+1}$ , and  $F_n(t) = 1$  for  $t \geq t_n$  where  $n_i$  is the number of  $t_j$  that are smaller or equal to  $t_i$ . Further, let  $c$  be a clock with CDF  $F_c$ . Then the  $n$ -th KS statistic is given by

$$K_n \stackrel{\text{def}}{=} \sup_{t \in \mathbb{R}_0^+} |F_c(t) - F_n(t)|. \tag{3}$$

If the sample values  $t_1, \dots, t_n$  are truly drawn from the CDF  $F_x$ , then  $K_n \rightarrow 0$  almost surely as  $n \rightarrow \infty$  by the Glivenko-Cantelli theorem. Hence, for given  $\alpha$  and sample size  $n$ , we accept the hypothesis that the  $t_i$  were drawn from  $F_x$  iff  $K_n \leq K_{crit}/\sqrt{n}$ , where  $K_{crit}$  is a critical value given by the Kolmogorov distribution. Again, the critical values can be calculated or found in tables.

*Example 3.* The left-hand side of Fig. 4 shows a tiny example specification IOSA with clocks  $x$  and  $y$ . The expiration times of both are uniformly distributed with different parameters. The right-hand side depicts a sample from this IOSA. There are two steps to assess whether the observed data is a truthful sample of the specification with a confidence of  $\alpha = 0.05$ : (1) find a trace distribution that minimises the  $\chi^2$  statistic and (2) evaluate two KS tests to assess whether the observed time data is a truthful sample of  $\text{UNI}[0, 2]$  and  $\text{UNI}[0, 3]$ , respectively.



**Fig. 4.** Tiny example implementation IOSA and sample observation.

There are two classes of traces solely based on the action signature: ID 1–8 with **a!** and ID 9–14 with **b!**. Let  $p$  be the probability that a scheduler assigns to taking the left branch in  $l_0$  and  $1 - p$  that assigned to taking the right branch. Drawing a sample of size  $m$ , we expect  $p \cdot m$  times **a!** and  $(1 - p) \cdot m$  times **b!**. The empirical  $\chi^2$  value therefore calculates as

$\chi^2 = (8 - 14 \cdot p)^2 / (14 \cdot p) + (6 - 14 \cdot (1 - p))^2 / (14 \cdot (1 - p))$ , which is minimal for  $p = 8/14$ . Since it is smaller than  $\chi_{crit}^2 = 3.84$ , we found a scheduler that explains the observed frequencies.

Let  $t_1 = 0.26, \dots, t_8 = 1.97$  be the data associated with clock  $x$  and  $t'_1 = 0.29, \dots, t'_6 = 2.74$  be the data associated with clock  $y$ .  $D_8 = 0.145$  is the maximal distance between the empirical step function of the  $t_i$  and  $\text{UNI}[0, 2]$ . The critical value of the Kolmogorov distribution for  $n = 8$  and  $\alpha = 0.05$  is  $K_{crit} = 0.46$ . Hence, the inferred measure is sufficiently close to the specification. The KS test for  $t'_i$  and  $\text{UNI}[0, 3]$  can be performed analogously.

The acceptance of both the  $\chi^2$  and the KS test results in the overall statistical acceptance of the implementation based on the sample data at  $\alpha = 0.05$ .

Our intention is to provide general and universally applicable statistical tests. The KS test is conservative for general distributions, but can be made precise [8]. More specialised and thus efficient tests exist for specific distributions, e.g. the Lilliefors test [17] for Gaussian distributions, and parametric tests are generally preferred due to higher power at equal sample size. The KS test requires a comparably large sample size, and e.g. the Anderson-Darling test is an alternative.

*Error propagation.* A level of significance  $\alpha \in (0, 1)$  limits type 1 error by  $\alpha$ . Performing several statistical experiments inflates this probability: if one experiment is performed at  $\alpha = 0.05$ , there is a 5% probability to incorrectly reject a true hypothesis. Performing 100 experiments, we expect to see a type 1 error 5 times. If all experiments are independent, the chance is thus 99.4%. This is the *family-wise error rate* (FWER). There are two approaches to control the FWER: *single step* and *sequential* adjustments. The most prevalent example for the first is Bonferroni correction, while a prototype of the latter is Holm's method. Both methods aim at limiting the *global* type I error in the statistical testing process.

## 4.2 Algorithm Outline

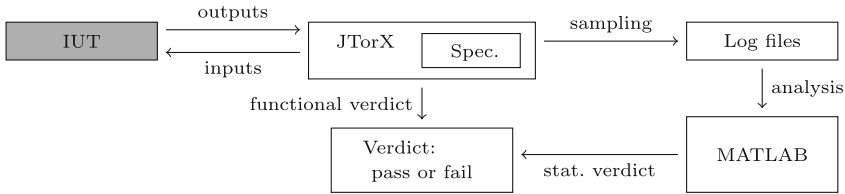
The overall practical procedure to perform MBT for *sa-ioco* is then as follows:

1. Generate an annotated test case  $\mathcal{T}$  of length  $k$  for the specification IOSA  $\mathcal{S}$ .
2. Execute  $\mathcal{T}$  on the IUT  $\mathcal{I}$   $m$  times. If the *fail* functional verdict is encountered in any of the  $m$  test executions then fail  $\mathcal{I}$  for functional reasons.
3. Calculate the number of KS tests and e.g. adjust  $\alpha$  to avoid error propagation.
4. Use SMT solving to find a scheduler s.t. the  $\chi^2$  statistic of the sample is below the critical value. If no scheduler is found, fail  $\mathcal{I}$  for probabilistic reasons.
5. Group all time stamps assigned to the same clock and perform a KS test for each clock. If any of them fails, reject  $\mathcal{I}$  for probabilistic reasons.
6. Otherwise, accept  $\mathcal{I}$  as conforming to  $\mathcal{S}$  according to  $\mathcal{T}$ .

*Threats to validity.* Step 5 has the potential to vastly grow in complexity if traces cannot be uniquely identified in the specification model. Recall Fig. 4 and assume  $\mathbf{a}! = \mathbf{b}!$ : it is now infeasible to differentiate between time values belonging to the left and to the right branch. To avoid this, we have to avoid this scenario at the time of modelling, or check *all possible combinations* of time value assignments.

## 5 Experiments

Bluetooth is a wireless communication protocol for low-power devices communicating over short distances. Its devices organise in small networks consisting of one *master* and up to seven *slave* devices. In this initialisation period, Bluetooth uses a frequency hopping scheme to cope with interferences. To illustrate our framework, we study the initialisation for one master and one slave device. It is inherently stochastic due to the initially random unsynchronised state of the devices. We give a high level overview and refer the reader to [11] for a detailed description and formal analysis of the protocol in a more general scenario.



**Fig. 5.** Experimental setup.

*Device discovery protocol.* Master and slave try to connect via 32 prescribed frequencies. Both have a 28-bit clock that ticks every  $312.5\ \mu\text{s}$ . The master broadcasts on two frequencies for two consecutive ticks, followed by a two-tick listening period on the same frequencies, which are selected according to

$$freq = [CLK_{16-12} + off + (CLK_{4-2,0} - CLK_{16-12}) \bmod 16] \bmod 32$$

where  $CLK_{i-j}$  marks the bits  $i, \dots, j$  of the clock and  $off \in \mathbb{N}$  is an offset. The master switches between two *tracks* every 2.56 s. When the 12th bit of the clock changes, i.e. every 1.28 s, a frequency is swapped between the tracks. We use  $off = 1$  for track 1 and  $off = 17$  for track 2, i.e. the tracks initially comprise frequencies 1-16 and 17-32. The slave scans the 32 frequencies and is either in sleeping or listening state. The Bluetooth standard leaves some flexibility w.r.t. the length of the former. For our study, the slave listens for 11.25 ms every 0.64 s and sleeps for the remaining time. It picks the next frequency after 1.28 s, enough for the master to repeatedly cycle through 16 frequencies.

*Experimental setup.* Our toolchain is depicted in Fig. 5. The IUT is tested on-the-fly via the MBT tool JTorX [2], which generates tests w.r.t. a transition system abstraction of our IOSA specification modelling the protocol described above. JTorX returns the functional *fail* verdict if unforeseen output or a timeout (quiescence) is observed at any time throughout the test process. We chose a timeout of approx. 5.2 s in accordance with the specification. JTorX's log files comprise the sample. We implemented the protocol and three mutants in Java 7:

- $\mathcal{M}_1$  Master mutant  $\mathcal{M}_1$  never switches tracks 1 and 2, slowing the coverage of frequencies: new frequencies are only added in the swap every 1.28 s.
- $\mathcal{M}_2$  Master mutant  $\mathcal{M}_2$  never swaps frequencies, only switching between tracks 1 and 2. The expected time to connect will therefore be around 2.56 s.
- $\mathcal{S}_1$  Slave mutant  $\mathcal{S}_1$  has its listening period halved: it is only in a receiving state for 5.65 ms every 0.64 s.

In all cases, we expect an increase in average waiting time until connection establishment. We anticipate that the increase leads to functional *fail* verdicts due to timeouts or to stochastic *fail* verdicts based on differing connection time distributions compared to the specification. We collected  $m = 100$ ,  $m = 1000$  and  $m = 10000$  test executions for each implementation, and used  $\alpha = 0.05$ .

**Table 1.** Verdicts and Kolmogorov-Smirnov test results for Bluetooth initialisation.

	Correct	Mutants		
	$\mathcal{M} \parallel \mathcal{S}$	$\mathcal{M}_1 \parallel \mathcal{S}$	$\mathcal{M}_2 \parallel \mathcal{S}$	$\mathcal{M} \parallel \mathcal{S}_1$
$k = 2$	<b>Accept</b>	<b>Reject</b>	<b>Accept</b>	<b>Accept</b>
$m = 100$	$D_m = 0.065$ $D_{crit} = 0.136$	—	$D_m = 0.110$ $D_{crit} = 0.136$	$D_m = 0.065$ $D_{crit} = 0.136$
Timeouts	0	40	0	0
$k = 2$	<b>Accept</b>	<b>Reject</b>	<b>Reject</b>	<b>Accept</b>
$m = 1000$	$D_m = 0.028$ $D_{crit} = 0.045$	—	$D_m = 0.05$ $D_{crit} = 0.045$	$D_m = 0.020$ $D_{crit} = 0.045$
Timeouts	0	399	0	0
$k = 2$	<b>Accept</b>	<b>Reject</b>	<b>Reject</b>	<b>Reject</b>
$m = 10000$	$D_m = 0.006$ $D_{crit} = 0.019$	—	$D_m = 0.043$ $D_{crit} = 0.019$	$D_m = 0.0193$ $D_{crit} = 0.0192$
Timeouts	0	3726	0	0

*Results.* Table 1 shows the verdicts and the observed KS statistics  $D_m$  alongside the corresponding critical values  $D_{crit}$  for our experiments. Statistical verdict **Accept** was given if  $D_m < D_{crit}$ , and **Reject** otherwise. Note that the critical values depend on the level of significance  $\alpha$  and the sample size  $m$ . The correct implementation was accepted in all three experiments. During the sampling of  $\mathcal{M}_1$ , we observed several timeouts leading to a functional *fail* verdict. It would also have failed the KS test in all three experiments.  $\mathcal{M}_2$  passed the test for  $m = 100$ , but was rejected with increased sample size.  $\mathcal{S}_1$  is the most subtle of the three mutants: it was only rejected with  $m = 10000$  at a narrow margin.

## 6 Conclusion

We presented an MBT setup based on stochastic automata that combines probabilistic choices and continuous stochastic time. We instantiated the theoretical

framework with a concrete procedure using two statistical tests and explored its applicability on a communication protocol case study.

## References

1. de Alfaro, L., Henzinger, T.A., Jhala, R.: Compositional methods for probabilistic systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 351–365. Springer, Heidelberg (2001). [https://doi.org/10.1007/3-540-44685-0\\_24](https://doi.org/10.1007/3-540-44685-0_24)
2. Belinfante, A.: JTorX: a tool for on-line model-driven test derivation and execution. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 266–270. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-12002-2\\_21](https://doi.org/10.1007/978-3-642-12002-2_21)
3. Bohnenkamp, H., Belinfante, A.: Timed testing with TorX. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 173–188. Springer, Heidelberg (2005). [https://doi.org/10.1007/11526841\\_13](https://doi.org/10.1007/11526841_13)
4. Briones, L.B., Brinksma, E.: A test generation framework for *quiescent* real-time systems. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 64–78. Springer, Heidelberg (2005). [https://doi.org/10.1007/978-3-540-31848-4\\_5](https://doi.org/10.1007/978-3-540-31848-4_5)
5. Cheung, L., Stoelinga, M., Vaandrager, F.: A testing scenario for probabilistic processes. *J. ACM* **54**(6), 29 (2007)
6. Cheung, L., Lynch, N., Segala, R., Vaandrager, F.: Switched PIOA: parallel composition via distributed scheduling. *Theor. Comput. Sci.* **365**(1), 83–108 (2006)
7. Cleaveland, R., Dayar, Z., Smolka, S.A., Yuen, S.: Testing preorders for probabilistic processes. *Inf. Comput.* **154**(2), 93–148 (1999)
8. Conover, W.J.: A Kolmogorov goodness-of-fit test for discontinuous distributions. *J. Am. Stat. Assoc.* **67**(339), 591–596 (1972)
9. D’Argenio, P.R., Lee, M.D., Monti, R.E.: Input/output stochastic automata. In: Fränzle, M., Markey, N. (eds.) FORMATS 2016. LNCS, vol. 9884, pp. 53–68. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-44878-7\\_4](https://doi.org/10.1007/978-3-319-44878-7_4)
10. Deng, Y., Hennessy, M., van Glabbeek, R.J., Morgan, C.: Characterising testing preorders for finite probabilistic processes. *CoRR* (2008)
11. Dufflot, M., Kwiatkowska, M., Norman, G., Parker, D.: A formal analysis of blue-tooth device discovery. *STTT* **8**(6), 621–632 (2006)
12. Eisentraut, C., Hermanns, H., Zhang, L.: On probabilistic automata in continuous time. In: LICS, pp. 342–351. IEEE Computer Society (2010)
13. Gerhold, M., Stoelinga, M.: Model-based testing of probabilistic systems. In: Stevens, P., Wasowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 251–268. Springer, Heidelberg (2016). [https://doi.org/10.1007/978-3-662-49665-7\\_15](https://doi.org/10.1007/978-3-662-49665-7_15)
14. Gerhold, M., Stoelinga, M.: Model-based testing of probabilistic systems with stochastic time. In: Gabmeyer, S., Johnsen, E.B. (eds.) TAP 2017. LNCS, vol. 10375, pp. 77–97. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-61467-0\\_5](https://doi.org/10.1007/978-3-319-61467-0_5)
15. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: FOSE, pp. 167–181. ACM (2014)
16. Hierons, R.M., Merayo, M.G., Núñez, M.: Testing from a stochastic timed system with a fault model. *J. Log. Algebr. Program.* **78**(2), 98–115 (2009)
17. Hollander, M., Wolfe, D.A., Chicken, E.: *Nonparametric Statistical Methods*. Wiley, Hoboken (2013)
18. Katoen, J.P.: The probabilistic model checking landscape. In: LICS. ACM (2016)

19. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. *Form. Methods Syst. Des.* **34**(3), 238–304 (2009)
20. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. *ACM* (1989)
21. Larsen, K.G., Mikucionis, M., Nielsen, B.: Online testing of real-time systems using UPPAAL. In: Grabowski, J., Nielsen, B. (eds.) *FATES 2004*. LNCS, vol. 3395, pp. 79–94. Springer, Heidelberg (2005). [https://doi.org/10.1007/978-3-540-31848-4\\_6](https://doi.org/10.1007/978-3-540-31848-4_6)
22. Milner, R. (ed.): *A Calculus of Communicating Systems*. LNCS, vol. 92. Springer, Heidelberg (1980). <https://doi.org/10.1007/3-540-10235-3>
23. Núñez, M., Rodríguez, I.: Towards testing stochastic timed systems. In: König, H., Heiner, M., Wolisz, A. (eds.) *FORTE 2003*. LNCS, vol. 2767, pp. 335–350. Springer, Heidelberg (2003). [https://doi.org/10.1007/978-3-540-39979-7\\_22](https://doi.org/10.1007/978-3-540-39979-7_22)
24. Segala, R.: *Modeling and verification of randomized distributed real-time systems*. Ph.D. thesis, Cambridge, MA, USA (1995)
25. Stoelinga, M.: *Alea jacta est: verification of probabilistic, real-time and parametric systems*. Ph.D. thesis, Radboud University of Nijmegen (2002)
26. Thrun, S., Burgard, W., Fox, D.: *Probabilistic Robotics*. MIT press, Cambridge (2005)
27. Tretmans, J.: Conformance testing with labelled transition systems: implementation relations and test generation. *Comput. Netw. ISDN Syst.* **29**(1), 49–79 (1996)
28. Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) *Formal Methods and Testing*. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-78917-8\\_1](https://doi.org/10.1007/978-3-540-78917-8_1)
29. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.* **22**(5), 297–312 (2012)



# Strategy Synthesis for Autonomous Agents Using PRISM

Ruben Giaquinta<sup>1</sup>, Ruth Hoffmann<sup>3</sup>, Murray Ireland<sup>2</sup>, Alice Miller<sup>1</sup>,  
and Gethin Norman<sup>1</sup>(✉)

<sup>1</sup> School of Computing Science, University of Glasgow, Glasgow, UK  
gethin.norman@glasgow.ac.uk

<sup>2</sup> School of Engineering, University of Glasgow, Glasgow, UK

<sup>3</sup> School of Computer Science, University of St Andrews, St Andrews, UK

**Abstract.** We present probabilistic models for autonomous agent search and retrieve missions derived from Simulink models for an Unmanned Aerial Vehicle (UAV) and show how probabilistic model checking and the probabilistic model checker PRISM can be used for optimal controller generation. We introduce a sequence of scenarios relevant to UAVs and other autonomous agents such as underwater and ground vehicles. For each scenario we demonstrate how it can be modelled using the PRISM language, give model checking statistics and present the synthesised optimal controllers. We conclude with a discussion of the limitations when using probabilistic model checking and PRISM in this context and what steps can be taken to overcome them. In addition, we consider how the controllers can be returned to the UAV and adapted for use on larger search areas.

## 1 Introduction

Autonomous vehicles such as unmanned aerial vehicles, autonomous underwater vehicles and autonomous ground vehicles have widespread application in both military and commercial contexts. Investment in autonomous systems is growing rapidly, the UK government is investing £100 million into getting driverless cars on the road, while the worldwide market for commercial applications of drone technology has been valued at over \$127 billion. For example, the U.S. Office of Naval Research has demonstrated how a swarm of unmanned boats can help to patrol harbours, the Defence Advanced Research Projects Agency has launched a trial of the world's largest autonomous ship and NASA has deployed Mars Rovers which, on receipt of instructions to travel to a specific location, must decide on a safe route.

Understandably, there are concerns about safety and reliability of autonomous vehicles. Recently researchers exposed design flaws in drones by deliberately hacking their software and causing them to crash [34], and US regulators discovered that a driver was killed while using the autopilot feature of a Tesla car due to the failure of the sensor system to detect another vehicle.



Incidents like these and the lack of design and analysis tools to prove system compliance under all nominal and adverse operating conditions are preventing regulatory bodies from issuing clear certification guidelines.

Autonomous agents almost always follow a variation of the same core process: *perception*, *cognition* and *actuation*. Perception is achieved through the system sensor suite, giving the agent a picture of the current environmental state. Actuation governs how the agent interacts with the environment and cognition is where the agent decides at run-time what goals to set and how to achieve them. A critical question is how to implement this decision-making process. Current best-practice uses a software *controller* that pre-determines the behaviour of the agent under a given set of internal parameter values and environmental conditions. However, *can controllers be generated automatically and in such a way as to ensure that the resulting behaviour is safe, efficient and secure under all conceivable operational scenarios and system failures?*

Guaranteeing reliability of autonomous controllers using testing alone is infeasible, e.g. [15] concludes that autonomous vehicles would need to be driven hundreds of billions of miles to demonstrate their reliability and calls for the development of innovative methods for the demonstration of safety and reliability that could reduce the infeasible burden of testing. Formal verification offers hope in this direction having been used both for controller synthesis and for verifying the reliability and safety of autonomous controller logic. In this paper we investigate the use of probabilistic model checking and the probabilistic model checker PRISM for automatic controller generation. Our ultimate goal is to develop software, based on the techniques described here that can be embedded into controller software to generate *adaptable* controllers that are *verified* to be *optimal*, *safe* and *reliable* by design. Specifically we:

1. describe PRISM models for a suite of scenarios inspired by situations faced by a range of autonomous agents;
2. present synthesised (optimal) controllers for the different scenarios and examine their performance;
3. discuss the limitations of this approach and the next steps to overcome them.

**Related Work.** There has been significant recent work using Markov decision processes, temporal logic specifications and model checking for generating controllers of autonomous systems. These works differ in the temporal logic specifications used and include approaches using the branching time logic PCTL [22, 36], linear time temporal logic LTL [7, 35], metric temporal logic [11], rewards [30] and multi-objective queries [21, 23]. Also, both partially observable Markov decision processes, e.g. [29, 31], and stochastic games, e.g. [8, 32] have been used in conjunction with temporal logic for controller synthesis of autonomous agents.

Formal verification of robot missions is considered in [27], however here the focus is on evaluating existing controllers. Similarly, in [6] model checking is used to verify the decision making aspect of autonomous systems. Model checking has also been used for analysing the behaviour of groups or swarms of autonomous agents including: agents in a pursuer-evader [2] scenario, foraging swarms [18, 24],

co-operative missions [13] and surveillance and convoy missions [5]. Concerning using formal verification to obtain certification of correctness, [33] uses (non-probabilistic) model checking to verify unmanned aircraft system controllers against the Civil Aviation Authority's regulations.

In previous work [12] we have presented a PRISM model of a UAV with a fixed controller searching for objects in a defined, gridded area following a fixed path. The real parameters are derived from a simulation model and the process of property verification using PRISM is compared to Monte Carlo simulation. This model is described in Scenario 1 (see Sect. 3).

## 2 Background

We now introduce Markov decision processes (MDPs) and probabilistic model checking of MDPs in PRISM. For any finite set  $X$ , let  $Dist(X)$  denote the set of discrete probability distributions  $X$ .

**Markov Decision Processes.** MDPs model discrete time systems that exhibit both nondeterministic and probabilistic behaviour.

**Definition 1.** A Markov decision process (MDP) is a tuple  $M = (S, \bar{s}, A, P)$  where:  $S$  is a finite set of states and  $\bar{s} \in S$  is an initial state;  $A$  is a finite set of actions;  $P : S \times A \rightarrow Dist(S)$  is a (partial) probabilistic transition function, mapping state-action pairs to probability distributions over  $S$ .

In a state  $s$  of an MDP  $M$ , there is a nondeterministic choice between the *available* actions in  $s$ . These available actions, denoted  $A(s)$ , are the actions for which  $P(s, a)$  is defined. If action  $a$  is selected, then the successor state is chosen probabilistically, where the probability of moving to state  $s'$  equals  $P(s, a)(s')$ . An execution of an MDP is a path corresponding to a sequence of transitions of the form  $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots$ , where  $a_i \in A(s_i)$  and  $P(s_i, a_i)(s_{i+1}) > 0$  for all  $i \geq 0$ . Let  $FPaths_M$  denote the finite paths of  $M$  and  $last(\pi)$  denote the last state of any finite path  $\pi$ .

Reward structures model quantitative measures of an MDP which are accumulated when an action is chosen in a state.

**Definition 2.** A reward structure for an MDP  $M = (S, \bar{s}, A, P)$  is a function of the form  $R : S \times A \rightarrow \mathbb{R}_{\geq 0}$ .

To reason about the behaviour of an MDP, we need to introduce the definition of *strategies* (also called policies, adversaries and schedulers). A strategy resolves the nondeterminism in an MDP by selecting the action to perform at any stage of execution. The choice can depend on the history and can be made randomly.

**Definition 3.** A strategy of an MDP  $M$  is a function  $\sigma : FPaths_M \rightarrow Dist(A)$  such that  $\sigma(\pi)(a) > 0$  only if  $a \in A(last(\pi))$ .

Under a strategy  $\sigma$  of an MDP  $M$ , the nondeterminism of  $M$  is resolved, and hence its behaviour is fully probabilistic. Formally, it corresponds to an (infinite) state discrete time Markov chain (DTMC) and we can use a standard construction on DTMCs [17] to build a probability measure over the infinite paths of  $M$ .

**Property Specifications.** Two standard classes of properties for MDPs are *probabilistic* and *expected reachability*. For a given state predicate, these correspond to the probability of eventually reaching a state satisfying the predicate and the expected reward accumulated before doing so. The value of these properties depends on the resolution of the nondeterminism, i.e. the strategy, and we therefore consider optimal (minimum and maximum) values over all strategies.

**The Probabilistic Model Checker PRISM.** PRISM [19] is a probabilistic model checker that allows for the analysis of a number of probabilistic models including MDPs. Models in PRISM are expressed using a high level modelling language based on the Reactive Modules formalism [1]. A model consists of a number of interacting modules. Each module consists of a number of finite-valued variables corresponding to the module’s state and the transitions of a module are defined by a number of guarded commands of the form:

$$[\langle \text{action} \rangle] \langle \text{guard} \rangle \rightarrow \langle \text{prob} \rangle : \langle \text{update} \rangle + \dots + \langle \text{prob} \rangle : \langle \text{update} \rangle$$

A command consists of an (optional) action label, guard and probabilistic choice between updates. A guard is a predicate over variables, while an update specifies, using primed variables, how the variables of the module are updated when the command is taken. Interaction between modules is through guards (as guards can refer to variables of all modules) and action labels which allow modules to synchronise. Support for rewards are through reward items of the form:

$$[\langle \text{action} \rangle] \langle \text{guard} \rangle : \langle \text{reward} \rangle;$$

representing the reward accumulated when taking an action in a state satisfying the guard.

PRISM supports the computation of an optimal probabilistic and expected reachability values, for details on how these values are computed and the temporal logic that PRISM supports, see [10]. PRISM can also synthesise strategies achieving such optimal values. Such a strategy is represented as a list of (optimal) action choices for each state of the MDP under study, this list can then be fed back into PRISM to generate the underlying DTMC, and hence allow further analysis of the strategy. For details on strategy synthesis see [20].

### 3 Scenarios

We describe a number of scenarios relevant for autonomous agents and how PRISM has been used for verification and controller synthesis. Each scenario is inspired by realistic situations for a range of autonomous vehicle applications, e.g. border patrol using autonomous vehicles, exploration of unexplored terrain, and search and rescue operations. However, in each case we present a simplified scenario involving an *autonomous agent* involving search within a defined area. The PRISM model and property files for each scenario are available from [37].

**Scenario 1: Fixed Controller.** In [12] we introduced abstract PRISM models representing an agent searching for and collecting objects randomly placed in a grid. The models are based on a physical system, namely a quadrotor UAV in operation inside a small, constrained environment in the University of Glasgow’s Micro Air Systems (MAST) Laboratory, a cuboidal flight space with a motion capture system for tracking UAVs. Continuous Simulink simulation models have been developed so that the effect of altering various aspects (including the search strategy) can be investigated via Monte Carlo simulation [14]. The purpose was to investigate the viability of a framework for analysing autonomous systems using probabilistic model checking of an abstract model where quantitative data for abstract actions is derived from small-scale simulation models.

The controller in this scenario is fixed and specifies that the agent searches the grid in a predetermined fashion, starting at the bottom left cell of the grid, travelling right along the bottom row to the bottom right cell, then left along the second row, and so on. The controller also specifies that if an object is found during search, then the agent attempts to pick up the object and, if successful, transports it to a specified deposit site. Whenever the agent’s battery level falls below a specified threshold, it returns to the base to recharge and once the battery is charged resumes the search. In both cases, search resumes from the previous cell visited, until all objects have been found or because the search can not continue (e.g. due to an actuator fault or the mission time limit has been reached).

We used MDP models and PRISM to analyse this scenario with a grid size of  $7 \times 4$  and either 2 or 3 objects. Although the controller is fixed, nondeterminism is used to represent uncertainty in the environment, specifically the time taken for the agent to execute actions, which were obtained from our small-scale simulation models. The PRISM models contain modules for the agent’s behaviour, movement, time and battery level, and objects. To reduce the size of the state-space, rather than encoding the random placement of the objects within the model, we develop a model where objects have fixed coordinates and consider each possible placement of the objects. For example, in the case of two objects there are 378 different possible placements for the objects and each model with fixed placement has approximately 200,000 states. To obtain quantitative verification results for the model where objects are randomly placed we perform multiple verification runs by considering each possible placement of the objects and take an average.

In the remainder of the section we synthesise optimal *controllers* for different scenarios with respect to the mission time. We achieve this using PRISM to encode the choices of the controller using nondeterminism. We remove the nondeterminism corresponding to environmental factors, e.g. the time taken to perform actions as these are not choices of the controller. By moving to stochastic games [28] we could separate the controller’s choices from that of the environment. However, implementations of probabilistic model checking for such games, e.g. PRISM-games [4], do not currently scale to the size of models we consider.

**Scenario 2: Control of Recharging.** In this scenario we introduce choice as to when the battery is recharged. More precisely, recharging is no longer enforced when the battery reaches a pre-determined lower threshold as in Scenario 1, but can be performed nondeterministically at any time during search. We assume that positions of the objects are fixed and the agent explores the grid in the pre-determined fashion described for Scenario 1 above.

**Table 1.** Scenario 2: performance of optimal and Scenario 1 controllers ( $7 \times 4$  grid).

Base	Depot	Object 1	Object 2	Expected mission time		Expected no. of battery charges		Probability of mission success	
				Scenario 1	Optimal	Scenario 1	Optimal	Scenario 1	Optimal
(0,0)	(2,2)	(3,3)	(4,3)	285.8	138.0	2.301	2.021	0.949	0.975
(0,0)	(2,2)	(1,1)	(4,3)	252.2	147.3	2.181	1.002	0.956	0.975
(0,0)	(6,3)	(2,1)	(4,2)	292.4	178.5	2.364	1.056	0.948	0.970
(1,2)	(6,3)	(3,0)	(5,1)	212.4	83.93	1.015	0.203	0.962	0.987
(3,2)	(6,3)	(2,1)	(4,2)	218.9	117.3	1.127	1.032	0.960	0.980
(6,3)	(0,0)	(2,1)	(4,2)	221.4	119.5	1.090	1.021	0.960	0.978

We use PRISM to find the minimum expected mission time and synthesise an optimal strategy that achieves this minimum for a suite of models involving two objects, varying the positions of the base, depot and objects. The synthesised strategies demonstrate that the optimal choice is to recharge when close to base, rather than waiting for the battery level to reach a threshold level. The performance of the synthesised optimal controller is compared to that of the fixed controller used in Scenario 1 (which recharges when the battery level reaches a threshold) in Table 1. The results demonstrate that the synthesised controller offers a significant performance improvement over the controller of Scenario 1. We see the expected mission time drastically reduces, the probability of a successful mission increases and the expected number of battery recharges decreases.

**Scenario 3: Control of Search.** We now generalise Scenario 2 to include control of the search path as well as recharging. Since allowing freedom of movement increases the complexity of our model, we focus on the search mode of the agent and abstract other modes (including take-off, hover and grab, see [14] for details).

Having the positions of the objects as constants in the PRISM model is not feasible if our aim is to generate optimal and realistic controllers as this means that the agent knows the locations of the objects it is searching for. In such a situation, the optimal search strategy is clear: go directly to the objects and collect them. We initially considered using partially observable MDPs (POMDPs) and the recent extension of PRISM [26]. Using POMDPs we can hide the positions of the objects and synthesise an optimal controller, e.g. one that minimises the

```

// number of unexplored cells
formula n = gp0+gp1+gp2+gp3+gp4+gp5+gp6+gp7+gp8+gp9+gp10+gp11;
// probability of finding object in an unexplored cell
formula p = objs/n;

```

**Fig. 1.** PRISM code: probability of finding an object in an unexplored cell.

```

// move east
[east] s=0 & gp=0 & gp0=0 & posx<X → (posx'=posx+1);
[east] s=0 & gp=0 & gp0=1 & posx<X & p≤1 → p : (s'=1)&(posx'=posx+1)&(gp0'=0)
      + 1-p : (posx'=posx+1)&(gp0'=0);
// move west
[west] s=0 & gp=0 & gp0=0 & posx>0 → (posx'=posx-1);
[west] s=0 & gp=0 & gp0=1 & posx>0 & p≤1 → p : (s'=1)&(posx'=posx-1)&(gp0'=0)
      + 1-p : (posx'=posx-1)&(gp0'=0);
// move north
[north] s=0 & gp=0 & gp0=0 & posy<Y → (posy'=posy+1);
[north] s=0 & gp=0 & gp0=1 & posy<Y & p≤1 → p : (s'=1)&(posy'=posy+1)&(gp0'=0)
      + 1-p : (posy'=posy+1)&(gp0'=0);
// move south
[south] s=0 & gp=0 & gp0=0 & posy>0 → (posy'=posy-1);
[south] s=0 & gp=0 & gp0=1 & posy>0 & p≤1 → p : (s'=1)&(posy'=posy-1)&(gp0'=0)
      + 1-p : (posy'=posy-1)&(gp0'=0);

```

**Fig. 2.** PRISM commands: searching cell with gridpoint 0.

expected mission time. However, we found the prototype implementation did not scale as it implements only basic analysis techniques.

Subsequently we investigated modelling hidden objects with MDPs. This was found to be feasible by monitoring the unexplored cells and using the fact that the probability of an object being found in a cell that has not been explored is  $obj/n$  where  $obj$  is the number of objects still to be found and  $n$  the number of unexplored cells. In an  $M \times N$  grid we associate the cell with coordinates  $(x, y)$  the integer (or *gridpoint*)  $x + y \cdot M$ . Before we give the PRISM code for an agent searching, we list the variables, formulae and constants used in the code:

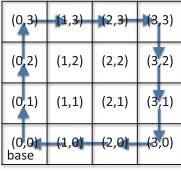
- variable  $s$  is the state of the agent taking value 0 when searching and 1 when an object has been found;
- variables  $posx$  and  $posy$  are the current coordinates of the agent and formula  $gp$  returns the corresponding gridpoint;
- constants  $X$  and  $Y$  represent the grid size, where  $X = M - 1$  and  $Y = N - 1$ ;
- variable  $gpi$  for  $0 \leq i \leq (X + 1) \times (Y + 1) - 1$  equals 1 when cell with gridpoint  $i$  has not been visited, and 0 otherwise;
- variable  $objs$  represents the number of objects yet to be found.

We assume the base and depot are fixed and located at position  $(0, 0)$ .

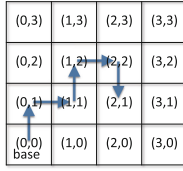
Figures 1 and 2 give the PRISM code extracts relevant for finding an object for a grid with 12 cells when the agent is searching the cell with gridpoint 0. To search the cell the agent needs to be searching and located in the cell ( $s = 0$  and  $gp = 0$ ). If the cell has already been searched ( $gp0 = 0$ ), then there is just a nondeterministic choice as to which direction to move. If the cell has not been

**Table 2.** Scenario 3: model checking results.

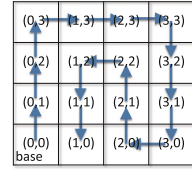
Grid size	No. of objects	Battery capacity	States	Transitions	Min expected mission time	Verification time (s)
$4 \times 4$	1	24	403,298	1,016,387	24.00	1.248
$4 \times 4$	2	24	860,689	2,212,391	38.60	2.840
$5 \times 4$	1	28	5,332,892	13,942,821	29.20	14.94
$5 \times 4$	2	28	11,841,031	31,526,709	46.27	38.77
$6 \times 4$	1	32	64,541,199	172,990,992	34.33	197.4
$6 \times 4$	2	32	149,723,921	408,008,297	53.94	514.5



(a) path 1 and recharge



(b) path 2 after recharge



(c) path (no recharge)

**Fig. 3.** Scenario 3: optimal controllers for  $4 \times 4$  grid, battery capacities 24 and 28.

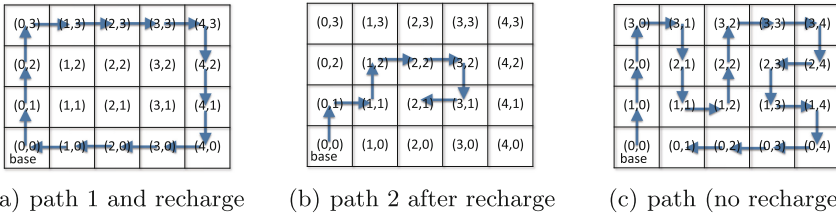
searched ( $gp\theta = 1$ ), then each choice includes the probability of finding an object using the formula in Fig. 1. The guard  $p \leq 1$  prevents PRISM reporting modelling errors due to potentially negative probabilities. Boundaries are encoded in guards rather than using knowledge of the grid, e.g. it is not possible to move south or west in gridpoint 0, to allow automated model generation for different grids.

After an object has been found ( $s = 1$ ), the agent deposits it at the base and resumes search if there are more objects to find. Returning to base either to deposit or recharge is encoded by a single transition with time and battery consumption updated assuming the controller takes a shortest path to the base. This modelling choice is to reduce the state space. Also to reduce the state space, we add conditions to guards in the battery module to prevent the agent moving to a position from which it cannot get to base with the remaining battery power. For example, for a  $5 \times 4$  grid, two objects and a battery capacity of 28, together these modelling choices reduce the state space from 24,323,956 to 11,841,031.

We synthesised optimal strategies for the minimum expected mission time for grids of varying sizes and number of objects. Table 2 presents model checking results in which we have chosen the minimum battery size that allows for a successful mission for the given grid. Figures 3(a)–(b) and 4(a)–(b) present the optimal strategies when searching for a single object. The figures give the optimal search paths which require returning to base during the search to recharge the battery. By increasing the capacity of the battery, the optimal strategy does not need to recharge. Figures 3(c) and 4(c) present optimal strategies for this

situation. In each case, the time to return to base when the object is found must be taken into consideration as opposed to only the time it takes to search.

**Scenario 4: Control of Sensors.** In this scenario we extend the power of the controller: as well as choosing the search path and when to recharge it can decide whether the search sensors are in a low or high power mode. In the high power mode the agent can search a cell, while in the low power mode it is only possible to traverse the cell. The high power mode for search is expensive in terms of time and battery use and can be unnecessary, e.g. when travelling over previously explored cells or returning to base to deposit or recharge. Again we assume the base and depot are fixed and located at position  $(0,0)$ . The PRISM model for this scenario extends that for Scenario 3 as follows. A variable  $c$  is added to the agent module, taking value 0 and 1 when its sensors are in low and high power modes respectively. The (nondeterministic) choices of the controller are then extended such that when deciding the direction of movement it also decides the power mode of the sensors for traversing the next cell. To aid analysis of the synthesised strategies, the action labels for direction of search include the power mode of the sensors, e.g. *south1* corresponds to moving south and selecting high power mode and *west0* to moving west and selecting low power mode.



**Fig. 4.** Scenario 3: optimal controllers for  $5 \times 4$  grid, battery capacities 28 and 32.

The PRISM code extract in Fig. 5 gives commands for moving east from cell with gridpoint 0 based on those in Fig. 2 for Scenario 3. The first two commands consider the case where the agents sensors are in high power mode ( $c = 1$ ) and the cell is unexplored. In both cases, since the sensors are in high power mode and the cell is unexplored, the probability of finding an object is as for Scenario 3. The difference is that in the first command the sensors are switched to lower power mode, while in the second the sensors remain in high power mode. The third and fourth commands represent the case when the sensors are in lower power mode and the cell is unexplored. Since the sensors are in lower power mode, the cell remains unexplored and there is no chance of finding the object. The final two commands consider the case where the cell has been previously explored. The PRISM model is also updated so that the time passage and battery consumption reflects the sensor's current power mode.



```

// sensors in high power mode and cell unexplored
[east0] s=0 & c=1 & gp=0 & gp0=1 & posx<X & p≤1 →
  p : (s'=1)&(posx'=posx+1)&(gp0'=0)&(c'=0) + 1-p : (posx'=posx+1)&(gp0'=0)&(c'=0);
[east1] s=0 & c=1 & gp=0 & gp0=1 & posx<X & p≤1 →
  p : (s'=1)&(posx'=posx+1)&(gp0'=0)&(c'=1) + 1-p : (posx'=posx+1)&(gp0'=0)&(c'=1);
// sensors in lower power mode and cell unexplored
[east0] s=0 & c=0 & gp=0 & gp0=1 & posx<X → (posx'=posx+1)&(c'=0);
[east1] s=0 & c=0 & gp=0 & gp0=1 & posx<X → (posx'=posx+1)&(c'=1);
// cell already explored (does not matter the sensors power mode)
[east0] s=0 & gp=0 & gp0=0 & posx<X → (posx'=posx+1) & (c'=0);
[east1] s=0 & gp=0 & gp0=0 & posx<X → (posx'=posx+1) & (c'=1);
    
```

Fig. 5. PRISM commands: searching cell 0, moving east and switching sensors off/on.

Table 3. Scenario 4: model checking results.

Grid size	No. of objects	Battery capacity	States	Transitions	Min expected mission time	Verification time (s)
3 × 3	1	12	53,367	222,557	12.78	0.206
3 × 3	1	16	80,107	351,209	12.11	0.272
3 × 3	2	12	103,063	435,302	19.83	0.365
3 × 3	2	16	154,911	687,246	18.88	0.420
4 × 4	1	18	18,445,790	90,303,355	21.88	58.58
4 × 4	1	24	27,587,864	139,945,165	20.50	83.60
4 × 4	2	18	36,379,747	180,055,826	32.22	124.8
4 × 4	2	24	54,464,317	279,117,740	30.60	181.9

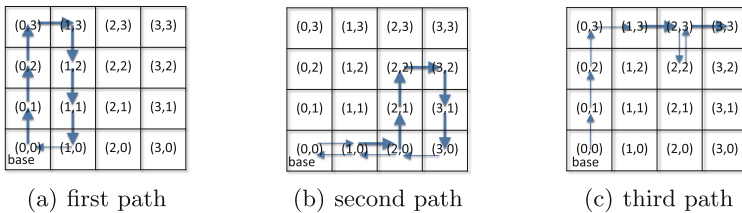


Fig. 6. Scenario 4: optimal controller for 4 × 4 grid, battery capacity 18 and one object.

Table 3 presents model checking results for this scenario including both those for the battery capacity from Scenario 3 (see Table 2) and for the minimum battery capacity required for a successful mission. Comparing with Table 2, allowing low and high power modes reduces the mission time, allows the mission to be completed with a smaller battery capacity and reduces recharging.

Comparing optimal strategies for Scenario 3 in Figs. 3 and 4 and those for Scenario 4 with the same battery capacity, the only difference is the low power mode is used when revisiting a cell. In Fig. 6 we present an optimal strategy for a 4 × 4 grid and battery capacity of 18. In this case, it is not feasible to complete the mission without using the lower power mode. Smaller arrows represent when the

sensors are in lower power mode. The move south during the third path before searching the final cell (3, 3) might not appear optimal. However, immediately before this step there is an equal chance of finding the object in the two remaining unexplored cells (2, 2) and (3, 3). By moving south after searching (2, 3) the time of returning to base is reduced when the object is found, at the cost of increasing the time to reach and search (3, 3) when the object is not found. In fact it is the case that initially moving east from (2, 2) also yield an optimal strategy, but was not the strategy synthesised by PRISM.

**Scenario 5: Control of Multiple Agents.** We now consider the case where there are multiple agents working together. We extend the PRISM model for Scenario 4 by having modules for two agents. In addition, since more than one cell can be explored at the same time, to simplify the PRISM code each cell is modelled as a separate module. The probability of finding an object is now dependent on both agents, and therefore we model this in a separate module.

```

// module for agent1
module agent1
  pos1x : [0..X] init basex; // x coordinate of agent1
  pos1y : [0..Y] init basey; // y coordinate of agent1
  // search (remain on grid and have sufficient battery)
  [search] u1=0 & u2=0 & pos1x<X & move_east → (pos1x'=pos1x+1); // east
  [search] u1=0 & u2=0 & pos1x>0 & move_west → (pos1x'=pos1x-1); // west
  [search] u1=0 & u2=0 & pos1y<Y & move_north → (pos1y'=pos1y+1); // north
  [search] u1=0 & u2=0 & pos1y>0 & move_south → (pos1y'=pos1y-1); // south
  // found object and not at base (go back to base)
  [end] (u1=1|u2=1) & !(agent1=base) → (pos1x' = basex) & (pos1y' = basey);
  // mission complete
  [end] (u1=1|u2=1) & agent1=base → true;
endmodule
// agent2 (rename agent1)
module agent2 = agent1[pos1x=pos2x, pos1y=pos2y, b1=b2] endmodule

```

**Fig. 7.** PRISM code: modules for *agent1* and *agent2* of Scenario 5.

The agent modules are presented in Fig. 7. Variables *pos1x* and *pos1y* represent the position of the first agent and *pos2x* and *pos2y* the second. Constants *basex* and *basey* give the position of the base and formula *base* the corresponding gridpoint. The search commands from the previous scenarios are modified and now synchronise on the action *search* with the gridpoint modules. Each command checks the variables *u1* and *u2* which indicate if an object has been found (see Fig. 7), since once the object is found, the agents return to base as the mission is complete. As the direction of movement is not encoded in the action *search*, preventing an agent moving in directions from which it cannot return to base with its remaining battery power is encoded in formulae *move\_east*, *move\_west*, *move\_north* and *move\_south* instead of the battery module.

For each cell in the grid there is a corresponding gridpoint module. The gridpoint modules for a  $3 \times 3$  grid are presented in Fig. 8. By using the constants

```

// constants used for renaming gridpoints
const int k0 = 0;
    :
const int k8 = 8;
// module for gridpoint 0
module gridpoint0
  gp0 : [0..1] init 1; // status of gridpoint0 (0 - explored and 1 - unexplored)
  // one of the agents searches the cell
  [search] (agent1=k0|agent2=k0) & (gp0=1) → (gp0'=0);
  // cell already searched or not being searched
  [search] !((agent1=k0|agent2=k0) & (gp0=1)) → true;
endmodule
// construct further gridpoints by renaming gridpoint0
module gridpoint1 = gridpoint0[gp0=gp1, k0=k1] endmodule
    :
module gridpoint8 = gridpoint0[gp0=gp8, k0=k8] endmodule

```

**Fig. 8.** PRISM code: module for gridpoints of Scenario 5.

```

// current gridpoint of agent1 and agent2 (derived from coordinates)
formula agent1 = pos1x+pos1y*(X+1);
formula agent2 = pos2x+pos2y*(X+1);
// retrieval probabilities module
module probabilities
  u1 : [0..1] init 0; // agent1 finds the object
  u2 : [0..1] init 0; // agent2 finds the object
  // agent1 and agent2 are searching different unexplored cells
  [search] s1=1 & s2=1 & agent1!=agent2 & 2*p≤1 → p : (u1'=1)
    + p : (u2'=1) + 1-2*p : true;
  // agent1 and agent2 searching the same unexplored cell
  // suppose each has the same chance of finding the object
  [search] s1=1 & s2=1 & agent1=agent2 & p≤1 → p/2 : (u1'=1)
    + p/2 : (u2'=1) + 1-p : true;
  // agent1 is searching unexplored cell while agent2 is not
  [search] s1=1 & s2=0 → p : (u1'=1) + 1-p : true;
  // agent2 is searching unexplored cell while agent1 is not
  [search] s1=0 & s2=1 → p : (u2'=1) + 1-p : true;
  // neither agent searching an unexplored cell
  [search] s1=0 & s2=0 → true;
endmodule

```

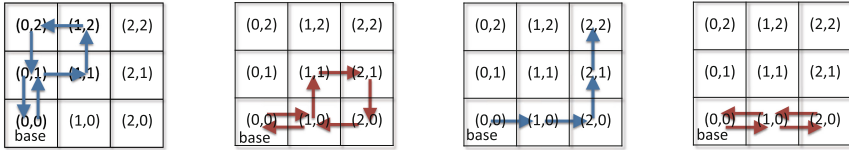
**Fig. 9.** PRISM code: retrieval probabilities module of Scenario 5.

$k_i$  we only need to explicitly construct the first gridpoint module and then use renaming. In the module for the first gridpoint (see Fig. 8), variable  $gp0$  is 1 when the cell is unexplored and 0 otherwise.

As stated above the probability of an agent finding an object is now a separate module, presented in Fig. 9. As before, the probability of an unexplored cell containing an object equals  $1/n$  where  $n$  is the number of unexplored cells (see Fig. 1). Formulae  $s1$  and  $s2$  evaluate to 1 if  $agent1$  and  $agent2$  are searching unexplored cells respectively. If the agents are searching different unexplored cells, then each agent has a chance of finding the object, but both cannot find the object as the object cannot be in two places at once.

**Table 4.** Scenarios 5 and 6: model checking results.

	Grid size	Battery capacity	States	Transitions	Min expected mission time	Verification time (s)
Scenario 5	3 × 3	16	53, 832	249, 588	12.00	0.285
	4 × 4	24	15, 555, 103	91, 859, 618	17.50	52.53
	5 × 4	28	293, 118, 691	1, 861, 895, 602	20.40	1,069
Scenario 6	3 × 3	16	153, 063	910, 392	11.93	0.701
	4 × 4	24	38, 165, 612	255, 234, 876	17.46	148.0
	5 × 4	28	691, 136, 157	4, 826, 058, 012	20.36	14,671



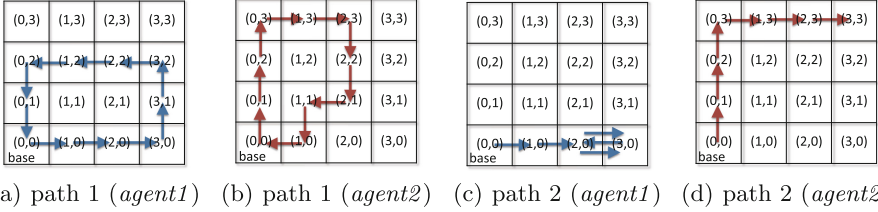
(a) path 1 (*agent1*) (b) path 1 (*agent2*) (c) path 2 (*agent1*) (d) path 2 (*agent2*)

**Fig. 10.** Scenario 5: optimal controller for 3 × 3 grid, battery capacity 16 and one object.

Table 4 presents model checking results for Scenario 5. As expected we see that searching with two agents can reduce the mission time over a single agent (see Table 2). Figures 10 and 11 present optimal strategies for grids of size 3 × 3 and 4 × 4. The optimal strategies are represented by the paths of the two agents before the object is found. As for the previous scenarios, as soon as the object is found the agents return directly to base. In both cases it is feasible for the agents to search the grid without recharging their batteries. However, this is not optimal due to the time required to return to base after finding the object. Neither the second path of *agent2* in Fig. 10 nor the second path of *agent1* in Fig. 11 contribute to the search. In both situations after recharging, there is only one cell to search ((2, 2) and (3, 3) respectively) and there is no gain in sending more than one of the agents to search this cell.

**Scenario 6: Control of Multiple Agents with Idle Mode.** As just discussed for Scenario 5, in certain situations there is no gain in both agents searching. For this reason in this scenario we add the ability for the controller to search using only one agent while the other idles at the base. Although this cannot reduce the mission time it can reduce power consumption and wear and tear.

Idling is introduced to the PRISM models through additional variables and the reward structure for time passage is updated to reduce the reward gained when an agent idles (see [37]). The optimal strategy will then choose idling over unnecessary movement, however as the reward is not reduced significantly it will use both agents to search when this can save time.



**Fig. 11.** Scenario 5: optimal controller for  $4 \times 4$  grid, battery capacity 24 and one object.

Table 4 includes model checking results for Scenario 6 (and 5). The reduced mission time from Scenario 5 to 6 is due to the change made to the reward structure and the generated optimal strategies yield the same expected mission time as those synthesised for Scenario 5. Figure 12 presents optimal strategies for a grid of size  $3 \times 3$ . This strategy is very different from that for Scenario 5 as in this case *agent1* searches the majority of the grid, while *agent2* searches only a small portion and returns to base and idles while *agent1* completes its search. For the  $4 \times 4$  grid the optimal strategy is initially the same as for Scenario 5 (see Fig. 11). However, in the second phase of the search there is no path for *agent1*, instead it idles at base while *agent2* searches the remaining cell.



**Fig. 12.** Scenario 6: optimal strategy for  $3 \times 3$  grid, battery capacity 16 and one object.

## 4 Conclusions and Future Work

We have demonstrated that probabilistic model checking and PRISM can be used for the synthesis of controllers for autonomous agents. However, there are clearly scalability issues as the models generated can have hundreds of millions of states for simple scenarios. Therefore, to analyse real-world applications, abstraction (and refinement) techniques are required. In particular, we will investigate using the game-based abstraction approach of [16] in this context, as well as symmetry reduction techniques [25] as there is symmetry both in the environment, e.g. in a grid structure, and between agents. Regarding the formal models and specifications, improving the efficiency of the POMDP implementation [26] could have significant modelling benefits, as in real applications control decisions must

be based only on the information from sensors, and therefore only on a partial view of the environment. Stochastic games are also required to model and separate the nondeterminism present in the environment from the choices of the controller. Combining these aspects will require the analysis of partially observable stochastic games which are harder to solve than POMDPs [3]. PRISM has support for multi-objective queries [9] and this will allow the synthesis of more specific controllers, e.g. that optimise the mission time while limiting both power consumption and failure, and ensuring safety requirements.

As for using PRISM for the analysis, the current way optimal strategies are exported can be improved. In particular, having a graphical representation would have simplified the analysis. In addition, allowing the analysis of a synthesised strategy directly would have saved considerable effort. Currently, to do this, the strategy has to be exported to a file and then imported back into PRISM (together with the state space and reward structures).

The PRISM models were developed from Simulink models [14]. PRISM generates the synthesised controllers as a lists of reachable states and optimal action choices for the states, this output from PRISM can be easily fed back into the *search* module of the Simulink models. Currently we are adapting the models for a larger search area consisting of adjoined discrete, symmetric regions. The same controller can then be used (modulo a symmetry transformation) on each region in turn until all objects have been located. At present the creation of UAV software from the Simulink code is automatic, future work will involve the direct embedding of the PRISM generated controllers into the UAV software.

**Acknowledgements.** This work was supported by EPSRC grant EC/P51133X/1. We would like to thank Dave Anderson and Euan McGookin for discussions on the autonomous systems that inspired this paper.

## References

1. Alur, R., Henzinger, T.: Reactive modules. *FMSD* **15**, 7–48 (1999)
2. Bohn, C.: Heuristics for designing the control of a UAV fleet with model checking. In: Grundel, D., Murphey, R., Pardalos, P., Prokopyev, O. (eds.) *Cooperative Systems. Lecture Notes in Economics and Mathematical Systems*, vol. 588, pp. 21–36. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-48271-0\\_2](https://doi.org/10.1007/978-3-540-48271-0_2)
3. Chatterjee, K., Doyen, L.: Partial-observation stochastic games: how to win when belief fails. *ACM Trans. Comput. Log.* **15**, 16 (2014)
4. Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: PRISM-games: a model checker for stochastic multi-player games. In: Piterman, N., Smolka, S.A. (eds.) *TACAS 2013. LNCS*, vol. 7795, pp. 185–191. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-36742-7\\_13](https://doi.org/10.1007/978-3-642-36742-7_13)
5. Choi, J.: Model checking for decision making behaviour of heterogeneous multi-agent autonomous system. Ph.D. thesis, Cranfield University (2012)
6. Dennis, L., Fisher, M., Lincoln, N., Lisitsa, A., Veres, S.: Practical verification of decision-making in agent-based autonomous systems. *ASE* **23**(3), 305–359 (2016)
7. Ding, X., Smith, S., Belta, C., Rus, D.: Optimal control of Markov decision processes with linear temporal logic constraints. *IEEE Trans. Autom. Control* **59**, 1244–1257 (2014)

8. Draeger, K., Forejt, V., Kwiatkowska, M., Parker, D., Ujma, M.: Permissive controller synthesis for probabilistic systems. *LMCS* **11**(2), 1–34 (2015)
9. Etesami, K., Kwiatkowska, M., Vardi, M., Yannakakis, M.: Multi-objective model checking of Markov decision processes. *LMCS* **4**, 1–21 (2008)
10. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D.: Automated verification techniques for probabilistic systems. In: Bernardo, M., Issarny, V. (eds.) *SFM 2011*. LNCS, vol. 6659, pp. 53–113. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-21455-4\\_3](https://doi.org/10.1007/978-3-642-21455-4_3)
11. Fu, J., Topcu, U.: Computational methods for stochastic control with metric interval temporal logic specifications. In: *Proceedings of CDC 2015* (2015)
12. Hoffmann, R., Ireland, M., Miller, A., Norman, G., Veres, S.: Autonomous agent behaviour modelled in PRISM – a case study. In: Bošnački, D., Wijs, A. (eds.) *SPIN 2016*. LNCS, vol. 9641, pp. 104–110. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-32582-8\\_7](https://doi.org/10.1007/978-3-319-32582-8_7)
13. Humphrey, L.: Model checking for verification in UAV cooperative control applications. In: Fahroo, F., Wang, L., Yin, G. (eds.) *Recent Advances in Research on Unmanned Aerial Vehicles*. LNCIS, vol. 444, pp. 69–117. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-37694-8\\_4](https://doi.org/10.1007/978-3-642-37694-8_4)
14. Ireland, M., Hoffmann, R., Miller, A., Norman, G., Veres, S.: A continuous-time model of an autonomous aerial vehicle to inform and validate formal verification methods. <http://arxiv.org/abs/1609.00177v1>
15. Kalra, N., Paddock, S.: Driving to safety: how many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transp. Res. Part A: Policy Pract.* **94**, 182–193 (2016)
16. Kattenbelt, M., Kwiatkowska, M., Norman, G., Parker, D.: A game-based abstraction-refinement framework for Markov decision processes. *FMSD* **36**, 246–280 (2010)
17. Kemeny, J., Snell, J., Knapp, A.: *Denumerable Markov Chains*. Springer, New York (1976). <https://doi.org/10.1007/978-1-4684-9455-6>
18. Konur, S., Dixon, C., Fisher, M.: Analysing robot swarm behaviour via probabilistic model checking. *Robot. Auton. Syst.* **60**(2), 199–213 (2012)
19. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) *CAV 2011*. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-22110-1\\_47](https://doi.org/10.1007/978-3-642-22110-1_47)
20. Kwiatkowska, M., Parker, D.: Automated verification and strategy synthesis for probabilistic systems. In: Van Hung, D., Ogawa, M. (eds.) *ATVA 2013*. LNCS, vol. 8172, pp. 5–22. Springer, Cham (2013). [https://doi.org/10.1007/978-3-319-02444-8\\_2](https://doi.org/10.1007/978-3-319-02444-8_2)
21. Lacerda, B., Parker, D., Hawes, N.: Multi-objective policy generation for mobile robots under probabilistic time-bounded guarantees. In: *Proceedings of ICAPS 2017* (2017)
22. Lahijanian, M., Andersson, S., Belta, C.: Formal verification and synthesis for discrete-time stochastic systems. *IEEE Trans. Autom. Control* **60**, 2031–2045 (2015)
23. Lahijanian, M., Kwiatkowska, M.: Specification revision for Markov decision processes with optimal trade-off. In: *Proceedings of CDC 2016*. IEEE (2016)
24. Liu, W., Winfield, A., Sa, J.: Modelling swarm robotic systems: a case study in collective foraging. In: *Proceedings of TAROS 2007* (2007)
25. Miller, A., Donaldson, A., Calder, M.: Symmetry in temporal logic model checking. *Comput. Surve.* **36**, 8 (2006)

26. Norman, G., Parker, D., Zou, X.: Verification and control of partially observable probabilistic systems. *Real-Time Syst.* **53**, 354–402 (2017)
27. O'Brien, M., Arkin, R.C., Harrington, D., Lyons, D., Jiang, S.: Automatic verification of autonomous robot missions. In: Brugali, D., Broenink, J.F., Kroeger, T., MacDonald, B.A. (eds.) *SIMPAR 2014*. LNCS (LNAI), vol. 8810, pp. 462–473. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-11900-7\\_39](https://doi.org/10.1007/978-3-319-11900-7_39)
28. Shapley, L.: Stochastic games. *Proc. Natl. Acad. Sci.* **39**, 1095–1100 (1953)
29. Sharan, R.: Formal methods for control synthesis in partially observed environments: application to autonomous robotic manipulation. Ph.D. thesis, California Institute of Technology (2014)
30. Soudjani, S., Majumdar, R.: Controller synthesis for reward collecting Markov processes in continuous space. In: *Proceedings of HSCC 2017*. ACM (2017)
31. Svoreňová, M., Chmelík, M., Leahy, K., Eniser, H., Chatterjee, K., Černá, I., Belta, C.: Temporal logic motion planning using POMDPs with parity objectives: case study paper. In: *Proceedings of HSCC 2015*. ACM (2015)
32. Svoreňová, M., Křetínský, J., Chmelík, M., Chatterjee, K., Cerna, I., Belta, C.: Temporal logic control for stochastic linear systems using abstraction refinement of probabilistic games. In: *Proceedings of HSCC 2015*. ACM (2015)
33. Webster, M., Fisher, M., Cameron, N., Jump, M.: Formal methods for the certification of autonomous unmanned aircraft systems. In: Flammini, F., Bologna, S., Vittorini, V. (eds.) *SAFECOMP 2011*. LNCS, vol. 6894, pp. 228–242. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-24270-0\\_17](https://doi.org/10.1007/978-3-642-24270-0_17)
34. Wilson, J.: Drones hacked and crashed by research team to expose design flaws. *Engineering and Technology* (2016)
35. Wolff, E., Topcu, U., Murray, R.: Robust control of uncertain Markov decision processes with temporal logic specifications. In: *Proceedings of CSC 2012*. IEEE (2012)
36. Yoo, C., Finch, R., Sukkariéh, S.: Provably-correct stochastic motion planning with safety constraints. In: *Proceedings of ICRA 2013*. IEEE (2013)
37. <http://www.prismmodelchecker.org/files/nfm18/>





# The Use of Automated Theory Formation in Support of Hazard Analysis

Andrew Ireland<sup>1</sup>(✉), Maria Teresa Llano<sup>2</sup>, and Simon Colton<sup>2,3</sup>

<sup>1</sup> School of Mathematical and Computer Sciences,  
Heriot-Watt University, Edinburgh, UK  
a.ireland@hw.ac.uk

<sup>2</sup> Department of Computing, Goldsmiths, University London, London, UK  
{m.llano,s.colton}@gold.ac.uk

<sup>3</sup> Games Academy, Falmouth University, Falmouth, Cornwall, UK

**Abstract.** Model checking and simulation are powerful techniques for developing and verifying the design of reactive systems. Here we propose the use of a complementary technique – automated theory formation. In particular, we report on an experiment in which we used a general purpose automated theory formation tool, HR, to explore properties of a model written in Promela. Our use of HR is constrained by meta-knowledge about the model that is relevant to hazard analysis. Moreover, we argue that such meta-knowledge will enable us to explore how safety properties could be violated.

**Keywords:** Formal methods · Verification · Hazard analysis

## 1 Introduction

Typically we have in mind a set of desired properties when we begin to develop a formal design model. Once constructed we verify our design model against the given properties. Here we propose a complementary approach to how the properties of a formal design model are obtained and used post-verification. Our starting point is SPIN and the Promela modelling language [6]. Firstly, we propose the use of a general purpose automated theory formation tool, HR<sup>1</sup> [2], to search for properties within Promela simulation traces. While such an approach will find properties that we expect of our models, it may also find properties of interest that we did not anticipate. Secondly, we propose an approach for discovering how a formally verified design could fail. That is, a formal counter-part to step 2 of Leveson’s STPA hazard analysis technique [8] where one considers how dysfunctional behaviour could emerge. Both aspects of our proposal rely on

---

The work reported here is funded by EPSRC Platform Grant EP/N014758/1. We thank the three anonymous NFM 2018 reviewers for their constructive feedback.

<sup>1</sup> HR is derived from the initials of the mathematicians Godfrey Harold Hardy and Srinivasa Aiyangar Ramanujan.

meta-knowledge to constrain the search for properties as well as dysfunctional behaviours. While meta-knowledge places an additional burden on the designer, we believe that it will deliver benefits during hazard analysis.

## 2 Background

HR was originally implemented as a system for *automated theory formation* (ATF) in domains of pure mathematics [2,4], e.g. the invention of integer sequences [3] and large-scale algebraic classification [11]. HR has also been used within the context of formal methods, specifically in discovering invariants from Event-B [1] animation traces [9]. HR forms theories about a domain through an iterative application of general purpose *production rules* (PRs) for concept invention. Each PR works by performing operations on the content of one or two input data tables – where a data table represents a concept by means of a set of examples. An application of a PR produces a new table, i.e. a new concept. HR then searches for relationships between the new concept and the concepts already in the theory. Specifically, it is looking to see if the new concept is: (i) equivalent to an existing concept; (ii) subsumed by or subsumes an existing concept; or (iii) empty. These relationships take the form of equivalence, implication, or non-existence conjectures, respectively.

To illustrate, we show in Fig. 1 the data tables used by HR to produce the concept of prime numbers. Thousands of PR applications occur during the ATF process, here we focus only on the specific PR applications that lead to the concept of prime numbers. Firstly, HR is given the concept of a divisor, as shown partially in Fig. 1 for integers from 1 to 10 ( $b|a$  where  $b$  is a divisor of  $a$ ). Secondly HR would apply the *size* PR with the parameterisation  $\langle 1 \rangle$  to count the number of tuples of each entry in column 1 (the Tau function table). HR then takes in this new concept and applies the *split* PR with the parameterisation  $\langle 2 = 2 \rangle$  to extract the entries in the previous data table whose value in the second column is 2. The resulting table defines the concept of a prime number.

Assuming the concept of non-square numbers has been formed previously by HR, with a data table formed by the examples  $\{2, 3, 5, 6, 7, 8, 10\}$  and the logical construction  $[a] : \neg(\exists b. (b|a \ \& \ b * b = a))$ . HR would then detect that the data table for the concept of non-square numbers subsumes the data table for the concept of prime numbers. That is, it sees that all of its prime numbers are also non-squares, and so conjectures that this is true for all prime numbers as follows:

$$\underbrace{2 = |\{b : b|a\}|}_{\text{prime number}} \rightarrow \underbrace{\neg(\exists b. (b|a \ \& \ b * b = a))}_{\text{non-square number}}$$

Because of the empirical nature of this process, false conjectures may be generated. Third party formal reasoning tools are employed to identify such false conjectures. Conversely, sometimes when developing a theory, conjectures which do not fully satisfy all the given examples may still be of interest. HR has a feature that allows such *near conjectures* to be identified, i.e. a user supplies a lower bound (percentage) on the number of examples that must be satisfied for a conjecture to be of interest.

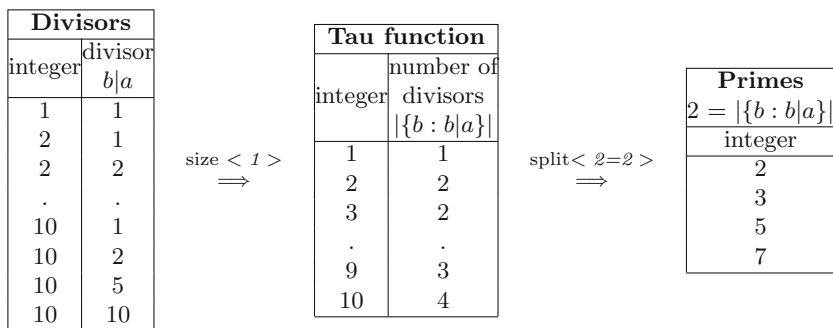


Fig. 1. Steps applied by HR to produce the concept of prime numbers.

### 3 Experiments with a Simple Design Model

For the purposes of our experiment we use a model of a laser control system. The laser is housed within a protective container which we refer to as the **Laser** unit. Access to the laser is via a door which is directly controlled by an **Operator**. In order to switch on the laser the **Operator** uses a **Control** unit. As well as controlling the power supply to the laser, **Control** also illuminates a warning light when the power is on. The model is both deliberately simple and poorly designed from the perspective of safety. Our Promela model of the system is shown in Fig. 2. There are two use cases an **Operator** can perform:

**PowerOn:** The **Operator** closes the **Laser** unit door then selects the power to be switched on. **Control** reacts by illuminating the power-on-light and then requests for power to be supplied to the **Laser**. The **Laser** unit then switches on the power and sends confirmation to **Control**.

**PowerOff:** The **Operator** selects the power to be switched off. **Control** reacts by requesting the **Laser** unit to stop supplying power to the laser. Once the **Laser** unit confirms the power is off, **Control** then switches off the power-on-light and the **Operator** opens the **Laser** unit door.

#### 3.1 Applying HR to the Laser Control System

In our experiment, the state variables that occur within the Promela model provide the basic concepts that are given to HR. As highlighted above, the simulation traces produced by SPIN represent the examples that HR requires in order to form conjectures. False conjectures can be identified using the SPIN model checker. One of the contributions of this work is an extension to HR that enabled it to discover temporal properties. Technically, *response properties* provided the greatest challenge and involved HR forming conjectures by relating two concepts as follows: if there is a state  $S_i$  for which the first concept has an example in the trace, then there is a state  $S_j$  for which the second concept has

```

bool opr_select_power_on  = false;
bool opr_door_closed     = false;
bool ctr_request_power_on = false;
bool ctr_power_on_light  = false;
bool lsr_power_on        = false;
bool lsr_confirm_power_off = true;

active proctype Operator(){
do
:: !opr_door_closed ->
   opr_door_closed = true; opr_select_power_on = true; ctr_power_on_light;
::  opr_select_power_on ->
   opr_select_power_on = false; !ctr_power_on_light; opr_door_closed = false;
od;}

active proctype Control(){
do
:: opr_select_power_on ->
   ctr_power_on_light = true; ctr_request_power_on = true;
   !lsr_confirm_power_off; !opr_select_power_on; ctr_request_power_on = false;
   lsr_confirm_power_off; ctr_power_on_light = false;
od;}

active proctype Laser(){
do
:: ctr_request_power_on ->
   lsr_confirm_power_off = false; lsr_power_on = true; !ctr_request_power_on;
   lsr_power_on = false; lsr_confirm_power_off = true;
od;}

```

Note that state variables associated with the `Operator` are prefixed by `opr`. Similarly, the prefixes `ctr` and `lsr` are associated with the `Control` and `Laser` units respectively.

**Fig. 2.** A Simple Laser Control System.

an example in the trace and  $j \geq i$ ; i.e. whenever the first concept happens, the second concept happens eventually. These conjectures are written in HR as:

$$\forall s_i . state(s_i) \wedge concept1(s_i, \dots) \rightarrow \exists s_j . state(s_j) \wedge concept2(s_j, \dots) \wedge j \geq i$$

where,  $state(s_x)$  refers to a step in the simulation trace occurring in time  $x$ , and  $concept(s_y, \dots)$  represent a tuple with the value of a concept in state  $s_y$ . Given that a simulation trace provides only a partial exploration of the state space, we have also extended HR to generate *near-response conjectures* – a natural generalization of the near-conjecture notion outlined in Sect. 2. In this way, we reduce the chances of missing properties that are rejected because a response property is violated by virtue of the finiteness of the traces given to HR.

### 3.2 Discovering Properties in Support of Hazard Analysis

HR will generate thousands of conjectures for a given simulation trace. As noted above we rely upon the designer to provide meta-knowledge about their models so that HR can constrain its search for interesting properties. Here we consider three kinds of meta-knowledge: Promela statements that represent: *hazards*, *defences* and *biddable* actions:

$$\begin{aligned} \langle \text{hazard} \rangle &= \{ \text{lsr\_power\_on} \} \\ \langle \text{defence} \rangle &= \{ \text{opr\_door\_closed} \} \\ \langle \text{biddable} \rangle &= \{ \text{opr\_door\_closed}, \text{opr\_select\_power\_on} \} \end{aligned}$$

Note that we adopt Reason’s [10] use of “defence” to denote mechanisms which prevent hazardous situations arising while “biddable”, inspired by Jackson’s work on Problem Frames [7], denotes actions performed by a human. In general we envisage an extensible meta-language for annotating model elements with respect to the role they play within a design and its context. Armed with this meta-knowledge we focus the search on three generic temporal properties.

Firstly, we search for safety conjectures of the form:

$$\Box(\langle \text{hazard} \rangle \rightarrow \langle \text{defence} \rangle) \quad (1)$$

That is, wherever a hazardous state is identified, as defined by the designer, then it must follow that a defence also holds. For our laser example HR generates 1565 implication conjectures. Using (1), and the meta-knowledge associated with the model, these are reduced to one conjecture:

$$\Box(\text{lsr\_power\_on} \rightarrow \text{opr\_door\_closed}) \quad (2)$$

This suggests that while power is being supplied to the laser there is only one defence in-place. Moreover, the defence *opr\_door\_closed* is also a member of the biddable set, i.e. the sole defence mechanism relies upon a human operator. Biddable operations are typically more vulnerable than say electrical or mechanical mechanisms involving redundancy. We will return to this point in Sect. 3.3. While (2) could have been anticipated by a designer, we would argue that the following properties are less obvious:

$$\Box(\text{lsr\_power\_on} \rightarrow \text{ctr\_power\_on\_light}) \quad (3)$$

$$\Box(\text{lsr\_power\_on} \rightarrow \neg \text{lsr\_confirm\_power\_off}) \quad (4)$$

Note the violation of (3) would not directly effect the safety of the system. However, if the power-on-light is not illuminated when the power is on then potentially the **Operator** could believe that it is safe to open the door, i.e. the violation of (3) could lead to the violation of (2). Property (4) ensures that **Control** is correctly informed of the laser’s status while power is being supplied. A violation of (4) would lead to **Control** incorrectly switching off the power-on-light with the potential violation of the high-level safety property described above. It is

worth noting that HR may find non-existence conjectures corresponding to (1) for a given model. The presence of such non-existence conjectures would provide a warning to the designer about the lack of defences within their model.

A second form of safety conjecture takes the form:

$$\Box!(\langle hazard \rangle \wedge !\langle defence \rangle) \quad (5)$$

Here we constrain HR to find conjectures where a hazard and the negation of a defence hold. Corresponding non-existence conjectures will strongly suggest safety properties. Here HR generates 20 non-existence conjectures. We follow a similar heuristic process as performed with the previous safety constraint. Firstly, we instruct HR to identify the non-existence conjectures that involve hazardous states. This narrows down the search to 10 conjectures. Next, we direct HR to further focus on conjectures that involve the negation of a defence, resulting in 5 conjectures. We end by removing non-existence conjectures that involve states that do not belong to the hazard or defence sets. The following instantiation of (5) is identified:

$$\Box!(l_{sr\_power\_on} \wedge !opr\_door\_closed)$$

Thirdly we search for response conjectures of the form:

$$\Box(\langle biddable \rangle \rightarrow \Diamond\langle hazard \rangle) \quad (6)$$

For our model of the laser control system, HR generates 22950 response conjectures and 18228 near-response conjectures (with lower threshold of 95%). Two near-response instantiations of (6) are given below with the percentage match:

$$\begin{aligned} \Box(opr\_select\_power\_on \rightarrow \Diamond l_{sr\_power\_on}) & 99.43\% \\ \Box(opr\_door\_closed \rightarrow \Diamond l_{sr\_power\_on}) & 97.93\% \end{aligned}$$

### 3.3 Breaking Properties

We now go beyond conventional verification, and consider how safety properties could be violated. Specifically, we focus on the safety property (2). There are a number of scenarios which could lead to the violation of (2). We consider here the simplest of scenarios. Given that `opr_door_closed` is a member of both the  $\langle defence \rangle$  and  $\langle biddable \rangle$  sets, then this is strongly suggestive of exploring a dysfunctional variate of the model in which `opr_door_closed` does not hold when it is supposed to hold. This dysfunctional behaviour can be included within the model by the addition of the following case to the definition of `Operator`:

```
:: !opr_door_closed ->
   opr_select_power_on = true; ctr_power_on_light;
```

SPIN will show that (2) is violated by the modified model. It is then the task of the designer to refine their design so as to mitigate for such an **Operator** error. In practice, scenarios that lead to single points of failure are typically not so simple. Part of our ongoing work is to further develop the idea outlined here. Specifically dealing with models where there are multiple defences and the application of a defence involves a chain of events.

## 4 Future Work and Conclusion

The experiments reported here demonstrate the potential for using meta-knowledge to guide HR in searching for properties relevant to hazard analysis. The version of HR we used is HR2. Running on a Macbook Pro (OS X Mavericks, processor 2.6 GHz Intel Core i5), HR2 could only deal with traces of 300 steps. As part of future work we aim to move to HR3 [5], which is significantly faster and more memory-efficient. This opens up the possibility of working with much larger datasets and exploring real-time theory formation. We chose to use HR2 for our initial experiments because of its GUI and support for browsing the results of the theory formation process. Such features will be added to HR3. The experiments reported here have also highlighted the need for a mechanism that will allow users to more easily tailor the conjecture making phase. Longer-term we envisage a computer-based *design assistant*, built upon HR3, which would run alongside a formal verification tool such as SPIN. The aim of such an assistant would be to help the designer explore the design space, as well as play a useful role during hazard analysis. What we propose compliments current practice, with the potential for identifying concerns which may otherwise be overlooked.

## References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Colton, S.: Automated Theory Formation in Pure Mathematics. Springer, Heidelberg (2002). <https://doi.org/10.1007/978-1-4471-0147-5>
3. Colton, S., Bundy, A., Walsh, T.: Automatic invention of integer sequences. In: Proceedings of AAAI (2000)
4. Colton, S., Muggleton, S.: Mathematical applications of inductive logic programming. Mach. Learn. **64**, 25–64 (2006)
5. Colton, S., Ramezani, R., Llano, T.: The HR3 discovery system: design decisions and implementation details. In: Proceedings of the AISB Symposium on Computational Scientific Discovery (2014)
6. Holzmann, G.J.: The SPIN Model Checker. Pearson Education, London (2003)
7. Jackson, M.: Problem Frames: Analysing and Structuring Software Development Problems. Addison-Wesley, Boston (2001)
8. Leveson, N.G.: Engineering a Safer World. MIT, Cambridge (2011)
9. Llano, M.T., Ireland, A., Pease, A.: Discovery of invariants through automated theory formation. Formal Aspects Comput. **26**, 203–249 (2011)
10. Reason, J.: Organizational Accidents Revisited. Ashgate, Farnham (2016)
11. Sorge, V., Meier, A., McCasland, R., Colton, S.: Automatic construction and verification of isotopy invariants. J. Autom. Reason. **40**(2–3), 221–243 (2008)



# Distributed Model Checking Using PROB

Philipp Körner<sup>(✉)</sup>  and Jens Bendisposto

Institut für Informatik, Universität Düsseldorf,  
Universitätsstr. 1, 40225 Düsseldorf, Germany

p.koerner@uni-duesseldorf.de, bendisposto@cs.uni-duesseldorf.de

**Abstract.** Model checking larger specifications can take a lot of time, from several minutes up to weeks. Naturally, this renders the development of a correct specification very cumbersome. If the model offers enough non-determinism, however, we can distribute the workload onto multiple computers in order to reduce the runtime.

In this paper, we present *distb*, a distributed version of PROB's model checker. Furthermore, we show possible speed-ups for real-life formal models on both a single workstation and a high-performance cluster.

## 1 Introduction

One way to verify software is explicit model checking, which checks a set of (invariant) predicates in every reachable state of a formal model. A simple model checking algorithm is shown in Algorithm 1: starting with a set of initial states, a directed graph is created according to the operations (aka state transitions) that the specification allows. It is checked whether a given property holds for each state, e.g., that there is no deadlock or that all invariant predicates are satisfied. Usually, open (aka unexplored) states are stored in a state queue and a set of visited nodes is stored in order to avoid checking the same state multiple times.

A big challenge however is the state space explosion problem: if we add more variables and operations to the model, the amount of states that need to be considered might grow exponentially.

One way to engage this issue is to add computational power and distributing the calculation of successor states and verification of invariants. Many formal models behave nondeterministically or have multiple initializations. If there is more than one state in the state queue, they can be distributed on multiple CPU cores or even workstations.

In this paper, we present the extension *distb* of PROB [21] that started as a distribution framework for Prolog but was tailored to overcome several challenges in the context of model checking. Now, *distb* allows checking industrial-sized specifications in a few hours that were impossible to check with vanilla PROB. *distb* is available for Linux and Mac OS X, but not for MS Windows. We focus on *distb*'s application for B [2] and Event-B [1].



---

**Algorithm 1.** Explicit State Model Checking Algorithm

---

**Require:** formal *specification* and function *desired-property*

```

results := ∅
queue := get-initial-states(specification)
seen := set(queue)
while queue ≠ ∅ do
  state := pop(queue)
  invariant-ok := check-invariant(specification, state)
  successors := compute-successors(specification, state)
  results := results ∪ desired-property(invariant-ok, successors)
  for s ∈ successors \ seen do
    enqueue(queue, s)
  end for
  seen := seen ∪ successors
end while
return results

```

---

### 1.1 B and PROB

The B specification language (which we will simply refer to as “B”) is part of the B-Method [2] developed by Jean-Raymond Abrial. The B-Method favors a “correct-by-construction” approach, where an abstract model is iteratively refined in order to end with a concrete implementation. B and its successor Event-B both work on a high level of abstraction and are based on set theory and first-order logic.

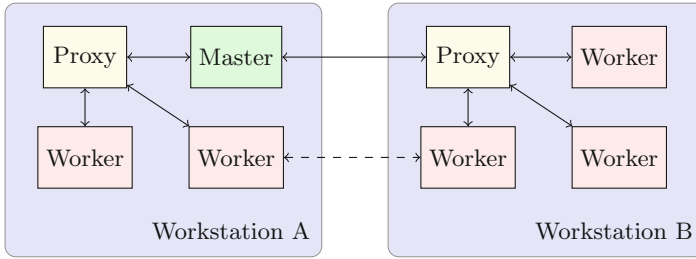
PROB [21] is an animator and model checker, initially for B, but now is capable of handling several formalisms, including Event-B, CSP-M, TLA<sup>+</sup> and Z. At its core, PROB implements a constraint solver to solve predicates in order to compute state transitions. PROB is implemented in SICStus Prolog [8].

## 2 Architecture Overview

*distb* is a distribution framework for Prolog, based on previous work in [6]. We focus on its application for PROB, i.e., distributed model checking. While PROB is able to verify temporal formulas, e.g., LTL, *distb* is only able to distribute invariant, deadlock and assertion checking. It is implemented in C with a small Prolog wrapper using SICStus Prolog’s foreign function interface. We also make use of the ZeroMQ library [14] which offers distributed messaging. While *distb* can handle any kind of computation task, we assume that they are tasks to verify a given state’s compliance to an invariant and computation of its successor states.

Starting with a root state, more states are generated and checked as the model checking process carries on. *distb* avoids to work on the same state multiple times as much as possible by storing hash codes of enqueued and processed states.

For *distb*, we opted for a master-worker architecture in order to match communication patterns offered by ZeroMQ. An architecture without a dedicated



**Fig. 1.** Typical setup in a distributed setting.

master, e.g., using MPI, is possible but also more complex. In the following, we give a simplified summary of each component’s tasks. We go into more detail in Sect. 3.

*The Master* oversees the entire model checking process. It monitors the distribution of work items and coordinates transfers between workstations. Furthermore, it collects and publishes checked and enqueued states. Once all states are checked, it sends a termination signal. In order to deal with infinite state spaces, the user may also limit the number of states.

*A Proxy* has some similar features to the master. Multiple proxies might be launched on the same workstation, but in most setups one proxy per machine suffices. Each worker on a workstation connects to exactly one proxy on the same computer. This is done by providing the same proxy ID to both the proxy and its workers. The proxy monitors the exact queue sizes and initiates transfers of work items between its assigned workers. Additionally, the proxy keeps the shared set of known and enqueued states, represented as a hash trie, up to date with the information provided by the master. The hash trie will be presented in Sect. 3.4 in detail. Moreover, it translates commands sent by the master into commands for the workers, e.g., sending work items to a worker assigned to another proxy, and forwards messages from its workers to the master.

*A Worker*, lastly, is the only component performing any work directly related to model checking. Each worker holds a local queue of states. Once it is not empty, a state is dequeued and checked (i.e., the invariant predicates are verified). Furthermore, the successors of this state are calculated and enqueued. Afterwards, a worker sends a package to the proxy, containing information about the processed state, its successors and additional statistics. Workers also periodically listen for commands sent by the proxy.

Each model checking process requires a single master. Each participating workstation should run at least one proxy. In order to initiate the calculation, at least one worker is required. A typical setup of a model checking process is shown in Fig. 1. There, we use two workstation, running two and three workers,

respectively. Arrows between components mean that there is direct communication between them, e.g., the master directly communicates with the proxies but not with workers. The dashed arrow, however, indicates that there might be direct communication, but the socket is closed after receiving an answer, i.e., workers communicate with each other at some time, but maintain no steady datastream they rely on.

### 3 Implementation

There are many subtle details challenging the implementation of *distb*. In the following, we state encountered problems as well as our proposals for solutions.

#### 3.1 Socket Patterns and Messages

*distb* uses ZeroMQ [14] to distribute the model checking work. ZeroMQ offers many useful communication patterns via different ZeroMQ socket types. We make use of the following three patterns:

- Publish-subscribe (PUB-SUB) allows sending messages from multiple sources to many subscribers, e.g., the master publishes commands to *all* proxies.
- Push-pull (PUSH-PULL) allows sending messages from many nodes to a sink (PULL socket). *distb* only uses one sink per connection, so, e.g., all proxies send (“push”) their results to a *single* master.
- Request-reply (REQ-REP) is the only bi-directional message pattern we employ. After a request is received, the reply will automatically be routed to the requesting component, e.g., a worker might send a request to share work with another worker, which in turn sends an acknowledgment as a reply.

In Table 1, we show which socket pattern is used between which components and what data is exchanged. Proxies request an ID from the master and workers request an ID from the proxy they connect to. These IDs are used in order to uniquely identify a component in certain commands, e.g., work balancing.

Sending states over network should be avoided due to bandwidth constraints. Instead, only hash codes of newly enqueued and checked states are transmitted (cf. Sect. 3.4). Workers push hash codes to their assigned proxy, which in turn pushes them to the master. The master distributes hash codes to all proxies.

All workers bind a TCP socket and always are able to receive work. Analogously, each component can connect to this socket in order to offer work. This way, workers share their queues with each other and the master can send work items from its own queue, e.g., the initial state.

#### 3.2 When is a Model Suitable for Distributed Model Checking?

*distb* is not a suitable tool for model checking all formal models. Naturally, *distb* cannot scale at all for sequential models, e.g., a simple counter with a single

**Table 1.** Socket types in *distb*

Master	Proxy	Worker	Messages and usage
REP	REQ		ID distribution
	REP	REQ	
PULL	PUSH		Hash codes and results (e.g., deadlock, invariant violation)
	PULL	PUSH	
PUB	SUB, PUB		Hash code propagation, sending commands (e.g., initiating global transfers, termination)
	PUB	SUB	
		REP	Receiving work
REQ	REQ	REQ	Sending work; connection is closed after transfer

initialization. There, the branching factor is one, i.e., each state only has one successor and there is only a single open node at most. In order to achieve best speed-ups, the state space should branch out in such way that many open nodes are available at all times.

As usual in distributed programming, adding more workers does not necessarily imply a bigger speed-up, e.g., using more workers than states in the state space does not provide any benefits. If some worker processes do not receive any work and stay idle, they might slow down the process overall due to the additional communication overhead involved.

Models that can be checked in very little time usually neither benefit from nor are hindered by adding more workers. If a state space is very large, *distb* may currently run out of memory quickly because all states are kept in main memory. Writing most states to disk and reading them back over time helps in order to delay this, usually by orders of magnitude. Obviously, a complete check of infinite models is unachievable in explicit state model checking. Instead, the number of states that should be considered has to be limited.

As it will be explained in more detail in Sect. 3.3, all states are serialized and deserialized. This additional overhead usually causes *distb* with a single worker to run slower than PROB. Thus, an optimal model for *distb* would feature only a small amount of variables which neither contain large nor nested data structures in order to minimize this overhead.

### 3.3 Passing States to C

PROB is implemented in Prolog and, thus, represents states as ordinary Prolog terms. However, terms passed into SICStus' foreign function interface only have a limited lifetime, i.e., exactly until the call that constructed the term returns. Afterwards, the memory on the stack is freed again. Creating a copy is not possible because it would end up on the same stack. If a reference to the list

of successor states was used as a return value, we could neither make use of SICStus' garbage collector nor free the memory ourselves.

Instead, we use an undocumented module named `fastrw` which offers a predicate `fast_buf_write` in order to serialize and another predicate `fast_buf_read` to deserialize a Prolog term. Both of these predicates work on a blob (binary large object) in a local buffer. However, this buffer can be accessed and duplicated with a simple call to `memcpy`.

Queue items reference such a blob as well as its size. Thus, when we call the function that processes a state, it has to `fast_buf_read` the state and `fast_buf_write` the successors. This blob also can be sent between multiple instances of SICStus.

We found that the overhead of serializing and deserializing states repeatedly is measurable and accumulates over time. However, the processing function of PROB usually is way slower in comparison. So far, we found no major issue and accepted the performance hit.

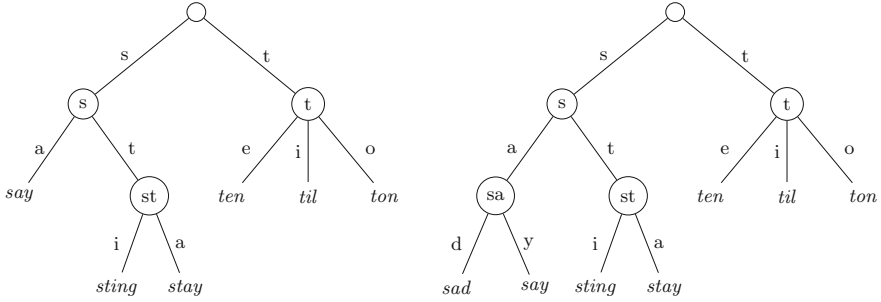
### 3.4 Visited States

In order to store whether states have been enqueued or checked already, we need a data structure that maps a state to the constants `ENQUEUED` and `PROCESSED`. However, keeping all seen states in memory is costly: each of them can be several megabytes in size. If a state space consists of only some thousand or a million of such states, it would be impossible to keep all of them in the main memory of an ordinary workstation. Thus, instead of the state itself we store its hash code. By default, a 160 bit SHA-1 hash is used. If we encounter states with the same hash code, we assume that the states are the same state. This can lead to unsoundness of the model checking if two different states produce the same hash value. An approximation for the probability  $p$  of a hash collision, given the number of possible keys  $d$ , and the number of stored keys  $n$  is [23]

$$p \approx 1 - e^{-n \left( \frac{n-1}{2d} \right)} \approx 1 - e^{-\left( \frac{n^2}{2d} \right)}$$

Since SHA-1 produces 160 bit hash values, the approximate collision probability for a billion elements is less than  $2^{-100}$ . For a trillion states it is less than  $2^{-80}$ . This trade-off lets us store an efficient fingerprint of the state but the chance of a collision for models that we can handle is about non-existent. Of course, it is possible to change the hash function to one that calculates a larger digest.

A good loading factor of a regular hash map however should be below ten per cent. This means that there still is a lot of overhead: more than nine times the payload if we inline the hash code as key or about more than four times the payload if we store 8 byte pointers instead of 20 byte hash codes. We found a more memory-efficient solution by adapting a variation of Phil Bagwell's Hash Array Mapped Tries [3]. We use a Trie (also referred to as a prefix tree) to store the states. We will refer to our implementation as digest trie. Knuth [16] defines tries as follows:



(a) Trie Containing the Words *say*, *strong*, *stay*, *ten*, *til*, *ton*

(b) Inserting *sad*

**Fig. 2.** An example for an 26-ary trie

“A trie is essentially an M-ary tree, whose nodes are M-place vectors with components corresponding to digits or characters. Each node on level  $\ell$  represents the set of all keys that begin with a certain sequence of  $\ell$  characters; the node specifies an M-way branch, depending on the  $(\ell + 1)$ st character.”

An example for a 26-ary trie<sup>1</sup> is shown in Fig. 2. Each branch represents a letter in the alphabet. We omit branches that have no successors. Then, looking up a word is simply following each letter until a leaf is reached and comparing the search term with the word found this way. For the word *sting* in Fig. 2a, one follows the branches *s*, *t* and *i* in this order and finds the word *sting*. Looking up *stand* fails because after following *s*, *t* and *a*, only *stay* is stored.

Inserting might be possible by simply adding a branch to the trie. A more complicated example is given in Fig. 2b, where we insert the word *sad*. An additional internal node needs to be added because the stored prefix *sa* for the contained word *say* collides with a prefix of *sad*.

The digest trie in *distb* is a 32-ary trie. In order to determine the next branch in the tree, 5 bits of the hash code are used. We try to ensure that the prefix tree has a relatively small depth in order to ensure a more performant lookup. Thus, the cryptographic hash function SHA-1 is used because its values are usually uniformly distributed.

**Shared Digest Trie.** In a naive implementation, each worker stores a copy of the digest trie. Duplicating the digest trie for multiple workers on the same workstation is very costly: a single copy usually takes up multiple gigabytes for larger models. Thus, we implemented a version [17] that resides in shared memory. While there has been work on a lock-free version of concurrent hash

<sup>1</sup> Strictly spoken, it should be a 27-ary trie including an end-of-word symbol in order to store both a word and one of its prefixes. Since the data we store has a fixed length, we omit this detail.

tries [22], the presence of a garbage collector for the shared memory is assumed. Our version of a shared digest trie allows multiple writers and readers to modify or read certain parts of the data structure concurrently. We use multiple locks to block access to certain parts of the trie.

The tree is partitioned into three different types of shared memory segments:

- meta information about the trie, i.e., how many internal nodes and hash codes are allocated as well as how many hash codes are marked as checked,
- an array of key-value pairs and
- an array of internal nodes of the trie.

While the first segment statically is of a fixed size, the other ones grow in size while the model checking process is running. Because resizing of shared memory segments is not possible in a UNIX-portable manner, we simply allocate more segments of a fixed size as needed.

Access to each of these segments is restricted differently: since the meta information only is accessed by the proxies and master, we use a single semaphore in order to coordinate read and write access.

For the key-value pairs, we use two semaphores: the first one is a counting semaphore allowing up to ten concurrent readers. This limit is chosen arbitrarily but matches current CPUs. The second one is a semaphore that manages write access. In order to gain write access, a process has to acquire the single write lock first before acquiring all read locks. Because there only is one write lock and all read locks are released eventually, deadlocks cannot occur.

Lastly, we use more fine-grained locking on the internal nodes. Each of the 32 sub-trees below the root node has such a combination of one counting semaphore to manage read access and one semaphore that manages write access. This means that each sub-tree can be read and written separately at the same time.

This technique has proven to be the best of multiple strategies we benchmarked in [17]. However, we argue that the amount of time spent processing a state is drastically larger than the amount of time spent in the hash trie. For our needs, any working locking strategy is good enough.

Note that the digest trie is used to store the set of known states, which is strictly growing. In particular, a delete operation is neither required nor implemented. Hash codes are written into the corresponding segment entirely before they are referenced in an inner node. Updating the status of the corresponding state (i.e., enqueued to checked) does not modify the hash code but only changes one bit. Thus, inconsistent reads of hash codes are impossible. In the worst case, a state cannot be found while it is added to the trie and is enqueued or checked again, which is sound behavior.

### 3.5 Work Sharing

When the model checking process begins, all workers start with an empty queue. The master will send the initial state to the first worker that is announced by a proxy. Once a worker accumulates enough work items, it is able to share some

of them. Of course, work queues may run empty during the process as well. Distributed model checking scales best when many queues are filled.

There are several strategies in order to distribute work: in some cases, the state space can statically be partitioned into (almost) disjunct parts. However, we do not do any static computations beforehand. A modulo-based approach assigns a state to a worker by calculating  $id = \text{hash}(state) \bmod \text{amount}(\text{workers})$ . One drawback of this approach is that many states have to be transferred between different workstations. As states can be several hundred megabytes in size, this would be too costly. Additionally, the amount of workers must not change. However, we want to be able to add and remove workstations on the fly depending on how well the model scales.

In order to oversee the work sharing, workers firstly send their queue sizes to their corresponding proxy. The proxy uses a *queue threshold* to classify queue sizes into one of three categories.

- The queue is empty and the worker should receive work items.
- The queue is not empty but below the queue threshold, usually a small value between 10 and 100 items. This worker should neither share nor receive work items. This is used in order to avoid many transfers at the beginning and end of the model checking process, when most queues are empty.
- The queue is above the queue threshold. This worker should share some of their work items if another one is empty.

This information is forwarded to the master as a queue fingerprint. In particular, exact queue sizes are not sent. An update is sent only when the fingerprint changes in order to reduce network traffic.

Proxies can initiate transfers between workers on the same workstation. Then, they flag the amount of workers as “in transfer” in their queue fingerprint. The master initiates transfers between workers on different workstations. However, local transfers are always favored over cross-workstation transfers.

For both kinds of the transfers, the worker that should share its queue is sent the IP address and port of the empty worker. Then, the sharing worker connects to the endpoint, sends part of their queue and disconnects after an acknowledgment. All workers bind a separate TCP socket in order to receive work.

### 3.6 Proxy

The proxy was, admittedly, introduced as a hack. As shown in Sect. 3.1, we use multiple sockets per component for different types of data flow. For smaller setups, it was possible to maintain multiple connections between the master and each worker. However, each socket requires a unique file handle. On many (shared) computation clusters, the amount of file handles per process is limited.

Nonetheless, we needed a component that runs once per workstation anyway: one that initially sets up the shared memory and blocks access until an initial consistent state is reached. As it turned out, having a single writer in a shared memory setting avoids many concurrency issues as well.



Furthermore, the proxy is able to take some responsibilities that can be done on the same workstation. An example is local balancing. Another is to handle information sent by the worker, e.g., queue size and logging into a shared resource. This eliminates some unnecessary network traffic entirely.

### 3.7 Bandwidth Reduction

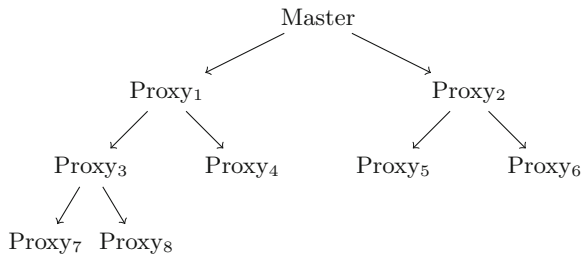
For some models, the distributed model checking scales wonderfully. This means, we can utilize hundreds of CPU cores which are under load and produce an enormous amount hash codes in a given time interval. We found that for some models, the master’s bandwidth does not suffice in order to provide each workstation with the hash updates.

This renders the master’s bandwidth to be the bottleneck of the computation resulting in many duplicates, meaning lots of useless work is done. Even though the model offers more potential for scaling, the entire process slows down if we add workers.

Inspired by streaming techniques in P2P networks, we implemented an application level multicast [25] for the hash codes. An example for a small setup consisting of eight workstations, each running a proxy, is shown in Fig. 3: there, the master only publishes hash codes to two proxies, which in turn propagate the information to two additional proxies each. Leaf nodes do not publish any information.

When a proxy joins the calculation, they are assigned a parent in the tree. Assume the  $n$ -th proxy connects to the master. Then, the master will include the endpoint of the  $\lfloor (n-1)/2 \rfloor$ -th proxy in the ID message and the joining proxy will connect its subscribing socket there. In case the calculated number is zero, the endpoint of the master’s hash publishing socket is provided instead.

By default, every proxy tries to publish hash codes. If there are no subscribers, ZeroMQ automatically drops the message. This allows the master to save bandwidth and to scale independently of the amount of participating workstations. The trade-off is increased latency of the hash code propagation. However, since we assume that we distribute the calculation in a local area network or cluster, the impact is neglectable.



**Fig. 3.** Hash code streaming tree. Arrows describe a “publishes to” relationship.

Additionally, in order to reduce overhead introduced by ZeroMQ messages, proxies put the hash codes they receive from workers immediately, the master may bundle multiple messages before propagating. The interval can be specified by the user. Our benchmarks ran with the default value that propagates hashes once every 25 ms. We found it works fairly well for all the models we tested. For models where states take a long time to check, this interval should be set to zero so that all currently available hash codes are propagated immediately in order to avoid checking states multiple times. In a setting where some duplicate checks are acceptable, e.g., when processing a single state is very fast, it might be increased even further. In general, this value should be fine-tuned according to the model.

## 4 Evaluation

In this section, we evaluate the performance of *distb* in two settings. Firstly, we consider a high-performance cluster where we can use multiple computation nodes and many hundred CPUs. All nodes used in this cluster have two Intel Xeon IvyBridge E5-2697, each of them consisting of 12 cores running at 2,70 GHz, offering up to 24 CPU cores per node. We reserved 100 GB RAM on each node. For network communication, we used standard 1 Gbit/s Ethernet. Each node runs a Red Hat 6.6 Linux. Secondly, we run the same version of *distb* on a single notebook with an Intel i7-7700HQ quad-core CPU and 16 GB RAM.

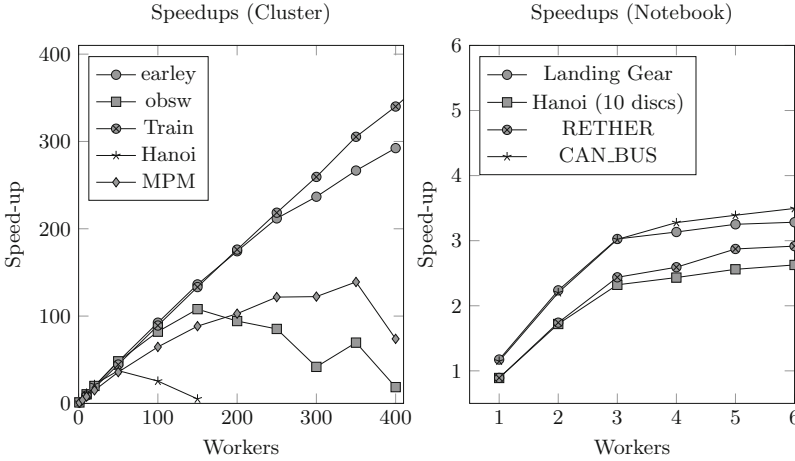
On the cluster, we run models that have a larger runtime, the smallest model takes about 30 min to model check with PROB. We could not check the largest model with PROB entirely thus far, although it checked half the state space in about three days.

When comparing the performance of *distb* with PROB, one has to keep in mind that *distb* suffers additionally to the distribution overhead from the fact that it has to serialize and deserialize all states. For larger states, this can be as expensive as verifying the invariant and calculating the successors. Thus, for some models, running *distb* with a single worker is much slower than PROB.

On the other hand, PROB usually does a little extra work compared to *distb*, e.g., it maintains the entire state space. Additionally, due to the different data structure, the lookup of seen states might be faster in *distb*. Thus, if the serialization is very fast, *distb* might be a bit faster than PROB. We have also noticed that if we add additional load to the CPU while running PROB, PROB speeds up a bit. A reason could be that Turbo Boost only gets activated if enough CPU load is present.

Speed-ups will always be given relative to PROB. For runtimes and speed-ups, we use the median value of ten repetitions. All models and their description are available at <https://github.com/pkoerner/distb>.

From Fig. 4 and Table 2, we can see that for suitable models with very long runtimes, like *Train* and *earley*, *distb* scales almost linearly even for hundreds of workers. Two smaller models that were developed by Space Systems Finland, *MPM* and *obsw* [9], take about half an hour with PROB and can be checked in less than half a minute given enough workers.



**Fig. 4.** Speed-ups in different configurations

**Table 2.** Runtimes (in seconds) and speed-ups on the high-performance cluster. †: estimated runtime, ‡: limited amount of initializations

earley (472886 states)	Workers	PROB	1	10	50	100	200	300	400	
	Runtime		25025.94	25280.63	2538.97	521.46	270.65	143.61	105.73	85.60
	Speed-up		1.00	0.99	9.86	47.99	92.47	174.26	236.70	292.36
obsw [9] (589279 states)	Workers	PROB	1	10	50	100	200	300	400	
	Runtime		2206.54	2021.54	212.96	45.82	26.84	23.40	52.75	118.45
	Speed-up		1.00	1.09	10.36	48.16	82.20	94.30	41.83	18.63
Train [1] (61648077 states)	Workers	PROB	10	50	100	200	300	400	500	
	Runtime		518400.00 †	58107.65	11649.45	5812.45	2942.85	1998.64	1524.78	1230.52
	Speed-up		1.00	8.92	44.50	89.19	176.16	259.38	339.98	421.29
Hanoi (14348909 states)	Workers	PROB	1	10	50	100				
	Runtime		17383.32	14107.04	1475.26	463.13	680.33			
	Speed-up		1.00	1.23	11.78	37.53	25.55			
MPM [9] (336649‡ states)	Workers	PROB	1	10	50	100	200	300	400	
	Runtime		1621.30	2114.86	209.26	45.27	25.08	15.78	13.26	21.93
	Speed-up		1.00	0.77	7.75	35.82	64.65	102.74	122.29	73.93

However, note that performance degrades in *distb* when too many workers are added. In our log files, we can see that messages get delayed for several seconds in the network. This could be caused by congestion of the internal switch that only has 20 Gbit/s throughput. Note that we ran multiple benchmarks in parallel and other users were active on the cluster at the same time. This would also explain the high variance in runtime we noted for these benchmarks, e.g., running the *obsw* model with 400 workers took between 39 s and 3.5 min, whereas the runtimes for 100 workers all were within 5 s.

We included a model of the Tower of Hanoi with 15 discs in our benchmarks because it has a particular property: the queue size blows up exponentially over time but collapses down to one possible state regularly, when there is only a new

single possible new position for the smallest disc. Most of the time, the overall queue size is relatively small and we did not expect *distb* to scale very well.

When we run smaller benchmarks on a quad-core notebook, *distb* usually scales linearly for three workers as can be seen in Fig. 4 and Table 3. This is due to the fourth core running both the proxy and master process. In particular, the proxy employs busy polling in order to react on input from multiple sources as soon as possible while the core logic still runs in a single thread. If more workers are added, minor additional speed-ups are gained due to hyper-threading.

As expected, the Tower of Hanoi model scales worst because it is the least suitable model for distribution of the ones benchmarked.

**Table 3.** Runtimes (in seconds) and speed-ups on a regular notebook.

Landing Gear (refinement 5) [11] (43307 states)	Workers	PROB	1	2	3	4	5	6
	Runtime	30.11	25.65	13.46	9.95	9.61	9.26	9.17
	Speed-up	1.00	1.17	2.24	3.03	3.13	3.25	3.28
Hanoi (10 discs) (59051 states)	Workers	PROB	1	2	3	4	5	6
	Runtime	33.97	38.15	19.75	14.63	13.96	13.27	12.93
	Speed-up	1.00	0.89	1.72	2.32	2.43	2.56	2.63
RETHEP protocol [24] (42254 states)	Workers	PROB	1	2	3	4	5	6
	Runtime	40.36	45.26	23.19	16.56	15.57	14.05	13.84
	Speed-up	1.00	0.89	1.74	2.44	2.59	2.87	2.92
CAN_BUS (John Colley) (132600 states)	Workers	PROB	1	2	3	4	5	6
	Runtime	73.85	64.20	33.54	24.46	22.53	21.78	21.13
	Speed-up	1.00	1.15	2.20	3.02	3.28	3.39	3.50

## 5 Related Work

Distributed model checking using PROB was also made available by integrating it with LTSMIN [4, 7]. For LTSMIN, states are split into several chunks, each containing only a single variable. This is done by multiple calls into the *fastrw* library (cf. Sect. 3.3 for more details). Thus, LTSMIN inherently runs slower without further optimizations.

However, LTSMIN offers a sophisticated caching mechanism: for each operation, states are projected into short states containing only the relevant variables. Then, PROB is only called if one of these variables changed. Otherwise, successor states are recombined from the cached value and the old state. For many models, this approach is very fast compared to PROB, trading reduced runtime for increased memory consumption. In a distributed setting, this concept does not scale as well as *distb*, because each worker maintains a separate cache. A preliminary evaluation of LTSMIN’s scaling behavior on a single machine can be found in [18].

Furthermore, the distributed version of LTSMIN allows checking LTL formulas with PROB, which *distb* is not yet capable of.

TLC is a model checker implemented in Java that offers both a parallel and distributed mode in order to check TLA<sup>+</sup> specifications [26]. Usually, all workers on a single workstation run inside the same JVM which allows sharing work with good performance. Furthermore, TLC offers a checkpointing mechanism that allows recovering progress after a graceful termination of the tool or after a crash. For its seen set, TLC stores hash codes only as well, however they are only 32-bits in size. Using the approximation from Sect. 3.4, the chance that *no* hash collision exists for one million states is less than  $10^{-50}$ . This almost guarantees that an unchecked state is discarded. Thus, we argue that such small fingerprints should not be used in order to verify larger state spaces.

While TLA<sup>+</sup> is a high-level formalism like B and Event-B, the input language for SPIN [15] is very low-level and its distributed version [19] tackles different issues. The main reason for distribution was not because of time but memory constraints: most of the main memory was used up by the hash table containing the visited states. The motivation was to distribute this hash table and, at this opportunity, to make use of additional computational power. Thus, the state space is partitioned beforehand in a way that relies on certain features of its input language.

When comparing SPIN with PROB, we notice that SPIN is able to deal with billions of states quite easily whereas PROB cannot cope with models that consist of more than a couple of million states. In [20], Leuschel has argued that these numbers are really difficult to compare due to the different level of abstractions. While the input language for SPIN is almost C like, the classical B language is almost pure mathematics. The high level of abstraction in B can lead to a significant reduction in the number of states because a single state at a high level of abstraction can represent hundreds or even thousand states in a low level language.

For distributed model checking, this has several consequences: firstly, *distb* usually is able to keep the entirety of the digest trie in main memory of each workstation. Secondly, PROB's states usually are larger and keeping all of them in main memory of a single workstation would be a hard issue. However, reading them from disk in sequence – like a queue – is relatively fast compared to random access of a hash map. Furthermore, states are inherently distributed on several workstations in the first place. Lastly, the computation of successors of a state takes a lot more time in PROB than in SPIN. Thus, the entire distribution overhead is rather small in comparison and allows more potential for scaling more easily.

## 6 Conclusion and Future Work

For suitable models, *distb* is able to achieve hundred-fold speed-ups compared to PROB. This renders it possible to model check medium-sized models in less than a minute instead of an hour. Furthermore, it allows model checking large specification that could not be handled by PROB before. We think for larger models like the *Train* benchmark, even better speed-ups are achievable, yet

would require InfiniBand instead of standard Ethernet. While the speed-ups we could achieve are very satisfactory, there still remains a list of features that are nice to have.

Storing states in their binary representation is costly. While SICStus Prolog reuses, e.g., atomic terms, and avoids their duplication, their blobs offer no structural sharing at all. Thus, keeping all blobs in memory often is not feasible for large models. In order to tackle this issue, we are currently evaluating storing these blobs on disk by making use of Google’s database LevelDB [10]. Reading from and writing to a HDD usually suffers from huge performance costs due to latency. Therefore, a user can specify an upper bound for the amount of blobs which are kept in memory. Once this upper bound is reached, additional states are written to disk asynchronously. Once no states reside in memory, the specified amount of states is read back from disk. This way, even large amounts of states can be queued without taking a major performance loss.

This should allow us to be able to check several billion B states of a very large model, a number we could not achieve thus far. Before, we were able to check about 160 million states of a *Hanoi* model but simply ran out of memory.

Additionally, we will try to reduce the serialization overhead. Many large models feature large constant values, e.g., a topology for a railway interlocking. Serialization and deserialization could be avoided if the value is replaced by a simple integer “handle”. The master could calculate all possible assignments for constants beforehand and provide a mapping from an integer to a set of constants to all workers. It will ensure that all workers share the same mapping which might not be the case due to, e.g., random enumeration.

Furthermore, we noticed that adding idle workers often slows down *distb*. We plan to add logic to the master or even a different tool that monitors queue sizes as well as progress that decides whether to hot-join additional workstations or to remove some of them from the calculation. Some models like the *Hanoi* example could benefit, where at the beginning most queues are empty but grow to large sizes as the model check progresses. This could be used, e.g., in a cloud settings where computational power can be added but, in order to reduce costs, may also be shut down if not needed.

Finally, *distb* does not make use of information about discharged proofs as described in [5]. Workers could send information which transition was used additionally to the hash code or only make use of it itself. This might lead to additional speed-ups because some invariants do not need to be verified at runtime if already proven beforehand.

Nonetheless, it would be interesting to compare the performance and scaling of *distb* with other state-of-the-art distributed model checkers, e.g., LTSMIN, TLC or SPIN. However, a fair comparison with SPIN would be hard because of different levels of abstraction. LTSMIN’s integration with PROB is usually inherently slower due to additional serialization and communication overhead. If caching is enabled, LTSMIN is usually faster but does not scale as good due to the fact that additional workers maintain their own caches. A proper comparison with TLC is feasible because it is possible to translate models freely between  $TLA^+$  and B [12, 13].

**Acknowledgement.** Computational support and infrastructure was provided by the “Centre for Information and Media Technology” (ZIM) at the University of Düsseldorf (Germany).

## References

1. Abrial, J.-R.: *Modeling in Event-B: System and Software Engineering*, 1st edn. Cambridge University Press, Cambridge (2010)
2. 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>
3. Bagwell, P.: *Ideal Hash Trees*. Es Grands Champs, vol. 1195 (2001)
4. Bendisposto, J., Körner, P., Leuschel, M., Meijer, J., van de Pol, J., Treharne, H., Whitefield, 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](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](https://doi.org/10.1007/978-3-642-10373-5_26)
6. Bendisposto, J.M.: *Directed and distributed model checking of B-specifications*. Ph.D. thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf (2015)
7. 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](https://doi.org/10.1007/978-3-642-14295-6_31)
8. Carlsson, M., Widen, J., Andersson, J., Andersson, S., Boortz, K., Nilsson, H., Sjöland, T.: *SICStus Prolog User’s Manual*, vol. 3. Swedish Institute of Computer Science Kista, Sweden (1988)
9. 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>
10. Ghemawat, S., Dean, J.: *LevelDB* (2011). <https://github.com/google/leveldb>
11. 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 Ameer, 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](https://doi.org/10.1007/978-3-319-07512-9_5)
12. Hansen, D., Leuschel, M.: Translating TLA<sup>+</sup> 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](https://doi.org/10.1007/978-3-642-30729-4_3)
13. Hansen, D., Leuschel, M.: Translating B to TLA<sup>+</sup> for validation with TLC. In: Ait Ameer, Y., Schewe, K.D. (eds.) *ABZ 2014*. LNCS, vol. 8477, pp. 40–55. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-662-43652-3\\_4](https://doi.org/10.1007/978-3-662-43652-3_4)
14. Hintjens, P.: *ZeroMQ: Messaging for Many Applications*. O’Reilly Media Inc., Newton (2013)
15. Holzmann, G.J.: The model checker SPIN. *IEEE Trans. Softw. Eng.* **23**(5), 279–295 (1997)
16. Knuth, D.E.: *The Art of Computer Programming: Sorting and Searching*, vol. III. Addison-Wesley, Boston (1973)

17. Körner, P.: Improving distributed model checking in ProB. Bachelor's thesis, Heinrich Heine Universität Düsseldorf, August 2014
18. Körner, P.: An integration of ProB and LTSmin. Master's thesis, Heinrich Heine Universität Düsseldorf, February 2017
19. Lerda, F., Sisto, R.: Distributed-memory model checking with SPIN. In: Dams, D., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, pp. 22–39. Springer, Heidelberg (1999). [https://doi.org/10.1007/3-540-48234-2\\_3](https://doi.org/10.1007/3-540-48234-2_3)
20. Leuschel, M.: The high road to formal validation: model checking high-level versus low-level specifications. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 4–23. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-87603-8\\_2](https://doi.org/10.1007/978-3-540-87603-8_2)
21. 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](https://doi.org/10.1007/978-3-540-45236-2_46)
22. Prokopec, A., Bagwell, P., Odersky, M.: Cache-aware lock-free concurrent hash tries. arXiv preprint [arXiv:1709.06056](https://arxiv.org/abs/1709.06056) (2017)
23. Sayrafiezadeh, M.: The birthday problem revisited. *Math. Mag.* **67**(3), 220–223 (1994)
24. 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)
25. Yeo, C.K., Lee, B.-S., Er, M.: A survey of application level multicast techniques. *Comput. Commun.* **27**(15), 1547–1568 (2004)
26. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA<sup>+</sup> 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](https://doi.org/10.1007/3-540-48153-2_6)





# Optimal Storage of Combinatorial State Spaces

Alfons Laarman<sup>(✉)</sup> 

Leiden University, Leiden, The Netherlands  
a.w.laarman@liacs.leidenuniv.nl

**Abstract.** Efficiently deciding reachability for model checking problems requires storing the entire state space. We provide an information theoretical lower bound for these storage requirements and demonstrate how it can be reached using a binary tree in combination with a compact hash table. Experiments confirm that the lower bound is reached in practice in a majority of cases, confirming the combinatorial nature of state spaces.

## 1 Introduction

Model checking has proven effective for automatically verifying correctness of protocols, controllers, schedulers and other systems. Because a model checker tool relies on the exhaustive exploration of the system’s *state space*, its power depends on efficient storage of states.

To illustrate the structure of typical states in model checking problems, consider Lamport’s Bakery algorithm in Fig. 1; a mutual exclusion protocol that mimics a bakery with numbering machine to prioritize customers. Due to limitation of computing hardware, the number is not maintained globally but reconstructed from local counters in  $\mathbb{N}[i]$  (for each process  $i$ ). For two processes, the state vector of this program consists of the two program counters ( $pc$ ) and all variables, i.e.  $\langle \mathbf{E}[0], N[0], pc_0, \mathbf{E}[1], N[1], pc_1 \rangle$ .<sup>1</sup> Their respective domains are:

$$\langle \{\top, \perp\}, [0 \dots 2], [0 \dots 7], \{\top, \perp\}, [0 \dots 2], [0 \dots 7] \rangle.$$

There are  $2 \times 3 \times 8 \times 2 \times 3 \times 8 = 2304$  possible state vectors. The task of the model checker is determine which of those are reachable from the initial state; here  $\iota \triangleq \langle \perp, 0, 0, \perp, 0, 0 \rangle$ . It does this using a next-state function, which in this case implements the semantics of the Bakery algorithm to compute the successor states of any state. For example, the successors of the initial state are:

$$\text{NEXT-STATE}(\langle \perp, 0, 0, \perp, 0, 0 \rangle) = \{ \langle \top, 0, 1, \perp, 0, 0 \rangle, \langle \perp, 0, 0, \top, 0, 1 \rangle \}$$

---

This work is part of the research programme VENI with project number 639.021.649, which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).



<sup>1</sup> We opt to order vectors as follows: variables and program counter of Process 0 ( $pc_0$ ), variables and program counter of Process 1 ( $pc_1$ ), etc. Section 6 discusses the effect of orderings.

```

bool E[2] = { false, false };
int N[2] = { 0, 0 };
void process(int i) { // with process id i = 0 or 1
0:   E[i] = true;
1:   N[i] = 1 + max(N[0], N[1]);
2:   E[i] = false;
#define j 0
3:   while (E[j]) { } // Wait until thread 0 receives its number
4:   while ((N[j] != 0) && ((N[j],j) < (N[i],i))) { }
#define j 1
5:   while (E[j]) { } // Wait until thread 1 receives its number
6:   while ((N[j] != 0) && ((N[j],j) < (N[i],i))) { }
/* begin critical section .. end critical section */
7:   N[i] = 0;
}

```

**Fig. 1.** Lamport’s “Bakery” mutual exclusion protocol for two threads. The wait loop is unrolled at Lines 4–7, where the process waits until all threads  $j$ , with smaller numbers or with the same number but with higher priority, expressed as  $(N[j], j) < (N[i], i)$ , passed their critical section. The boolean variable  $E[i]$  associated with process  $i$  serves to allow other threads to wait until  $i$  received a number in  $N[i]$ . For simplicity, we assume that each line can be executed atomically.

One successor represents the case where the first process executed Line 0; its program counter is set to 1 and  $E[0]$  is updated as a consequence. Similarly, the other successor represents the case where the second process executed Line 0.

While exhaustively exploring all reachable states, the model checker searches if it can reach a state from the set *Error*. For the Bakery algorithm with two threads, we have  $Error \triangleq \{ \langle b_0, n_0, 7, b_1, n_1, 7 \rangle \mid b_0, b_1 \in \{\top, \perp\}, n_0, n_1 \in [0 \dots 2] \}$ , i.e., all states where both processes reside in their critical section (= pc loc. 8). For completeness, Algorithm 1 shows the basic reachability procedure. The more states the reachability procedure can process, the more powerful the model checker, i.e., the larger program instances it can automatically verify. This number depends crucially on the size of the *visited states set*  $V$  in memory. Several techniques exist to reduce  $V$ : partial order reduction [19, 26], symmetry reduction [10, 30], BDDs [3, 7], etc. Here we focus on explicitly storing the states in  $V$  using state compression.

The potency of compression becomes apparent from two related observations:

**Locality.** Successors computed in the next-state function exhibit *locality*, e.g.,

$$\text{NEXT-STATE}(\langle \perp, 1, 4, \perp, 2, 6 \rangle) = \{ \langle \perp, 1, \mathbf{5}, \perp, 2, 6 \rangle, \langle \perp, 1, 4, \perp, 2, \mathbf{7} \rangle \}$$

---

**Algorithm 1.** The reachability procedure in a model checker.

---

**Data:**  $\iota$ , NEXT-STATE  
**Result:** {error, correct}

```

1  $V := \emptyset$ 
2  $Q := \{\iota\}$ 
3 while  $Q \neq \emptyset$  do
4    $Q := Q \setminus \{s\}$  for  $s \in Q$ 
5    $V := V \cup \{s\}$ 
6   for  $s' \in \text{NEXT-STATE}(s)$  do
7     if  $s' \notin V$  then
8       if  $s' \in \text{Error}$  then
9         return error
10       $Q := Q \cup \{s'\}$ 
11 return correct

```

---

Note that only program counters change value (marked bold in successors).

**Combinatorics.** Similar to the set of all possible state vectors, the set of reached state vectors is also highly *combinatorial*. Assuming  $\langle \perp, 1, 4, \perp, 2, 6 \rangle$  can be reached from the initial state  $\iota$ , we indeed saw four different vectors sharing large sub-vectors with their predecessors (underlined here):

$$\begin{aligned} \langle \perp, 0, 0, \perp, 0, 0 \rangle &\longrightarrow \langle \top, \underline{0}, 1, \underline{\perp}, 0, 0 \rangle, \langle \underline{\perp}, 0, 0, \top, \underline{0}, 1 \rangle \\ \langle \perp, 1, 4, \perp, 2, 6 \rangle &\longrightarrow \langle \underline{\perp}, \underline{1}, 5, \underline{\perp}, \underline{2}, 6 \rangle, \langle \underline{\perp}, \underline{1}, 4, \underline{\perp}, \underline{2}, 7 \rangle \end{aligned}$$

We hypothesize that the typical locality of the next-state function ensures that the set of reachable states exhibits this combinatorial structure in the limit. Therefore, storing each vector in its entirety in a hash table, would duplicate a lot of data. By folding the reachable state vectors in a tree, however, these shared sub-vectors only have to be stored once (more in Sect. 3).

In this paper, we investigate the lower bound on the space requirements of typical state spaces occurring in model checking. We do this by modeling the state spaces as an information stream. The values in this stream probabilistically depend on previously seen values, in effect modeling the locality in the next-state function. A simple application of Shannon’s information theory yields a lower bound for the storage requirements of our “state space stream”.

Subsequently, in Sects. 3 and 4, we investigate whether this lower bound can be reached in practice. To this end, we provide an implementation for the visited set  $V$ . A practical compressed data structure has as additional requirement that the query time, the time it takes to lookup and insert individual state vectors, is constant (or at least poly-logarithmic) in the length of the vector. The technique suggested by the information theoretical model, i.e., maintaining differences between successor states, does not satisfy this requirement. Therefore, we utilize a binary tree in combination with a *compact hash table*. By analyzing the best-case compression of this structure, we show that it indeed can reach information theoretical lower bound (at least in theory).

According to the same best-case analysis, our implementation of the ‘Compact Tree’ can compress up to tens of billions of large state descriptors (of tens to hundreds of integers) down to only one 32-bit integer per state. Extensive experimentation in Sect. 5 with diverse input models in four different input languages shows moreover that this compression is also reached in practice, and with little computational overhead.

## 2 An Information Theoretical Lower Bound

The fact that state spaces have combinatorial values is related to the fact that state generated by a model checker exhibit locality as we discussed in Sect. 1. *We will make no assumptions on the nature of the inputs, besides the locality of state generation.* In the current section, we will derive the *information entropy*—which is equal to the minimum number of bits needed for its storage—of a single state vector using basic notions from information theory.

*Information theory* abstracts away from the computational nature of a program by considering sender and receiver as black boxes that communicate data (signals) via a channel. The goal for the sender is to encode the data as small as possible, such that the receiver is still able to decode it back to the original. The encoded size depends on the amount of *entropy* in the data. In the most basic case, no statistical information is known about the data: each of  $X$  possible messages has an equal probability of taking one of its values and the entropy  $H$  is maximal:  $H(X) = \log_2(|X|)bit$ , i.e., the entropy directly corresponds to using one fixed-sized ( $\log_2(|X|)$ ) bit pattern for each possible message.

If more is known about the statistical nature of the information coming from the sender, the entropy is lower as more elaborate encodings can be used to reduce the number of bits needed per piece of information. A simple example is when we take into account the character frequency of the English language for encoding sentences. Assuming that certain characters are much more frequent, a code of fewer bits can be used for them, while longer codes can be reserved for infrequent characters. To calculate the entropy in this example, we need the probability of occurrence  $p(x)$  for each character  $x \in X$  in the English language. We can deduce this from analyzing a dictionary, or better a large corpus of texts. The entropy then becomes:  $H(X) = \sum_{x \in X} -p(x) \log_2(p(x))$

We apply the same principle now to state vectors. As data source, we use the next-state function to compute new states, as we saw in Sect. 1:

$$\text{NEXT-STATE}(\langle \perp, 1, 4, \perp, 2, 6 \rangle) = \{ \langle \perp, 1, \mathbf{5}, \perp, 2, 6 \rangle, \dots \}$$

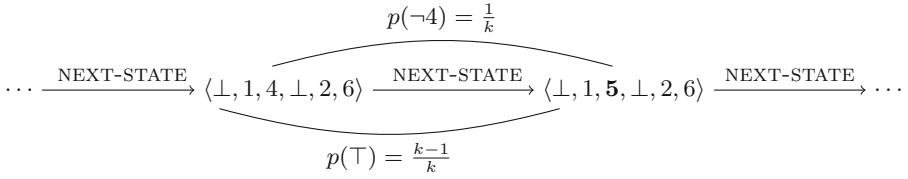
As a simplification, let states consist of  $k$  variables. By storing full states in the queue  $Q$ , the predecessor state is always known in the model checker’s reachability procedure (see  $s$  and  $s'$  on line 6 in Algorithm 1). Hence, we can abstract away from the one-to-many relation of the next-state function and instead view the arriving states as a  $k$ -periodic stream of variable assignments:

$$\langle v_0^0, \dots, v_{k-1}^0 \rangle, \langle v_0^1, \dots, v_{k-1}^1 \rangle, \dots, \langle v_0^{n-1}, \dots, v_{k-1}^{n-1} \rangle$$

It thus makes sense to describe the probability that a variable holds a certain value with respect to the same variable in the predecessor state: For each variable  $v_j^i$  with  $i \geq 0$  and  $0 \leq j < k - 1$ , both encoder and decoder can always look at the corresponding variable  $v_j^{i-1}$  in the predecessor to retrieve its previous value.

Since we are interested in establishing a *lower bound*, we may safely underestimate the number of variables changing value with respect to a state’s predecessor. It makes sense to assume that only one variable changes value, since with zero changes, the same state is generated (requiring no “new” space in  $V$ ). Hence, we take the following *relative* probabilities (see example Fig. 2):

$$p(v_j^i \neq v_j^{i-1}) = \frac{1}{k} \qquad p(v_j^i = v_j^{i-1}) = \frac{k-1}{k}$$



**Fig. 2.** The states generated with the NEXT-STATE function seen as a stream exhibiting locality. To derive a lower bound, we assume that locality changes only one value in each new vector, i.e., each vector that has to be stored. As there are  $k$  variables in the vector, the resulting probability that a variable changes is  $1/k$ . So the chance that it remains the same with respect to the predecessor is  $k^{-1}/k$ .

Let  $\langle d_0^0, \dots, d_{k-1}^0 \rangle, \langle d_0^1, \dots, d_{k-1}^1 \rangle, \dots, \langle d_0^{m-1}, \dots, d_{k-1}^{m-1} \rangle$ , be the domains of the state slots. As a simplification, we assume that all domains have  $u$  bits, resulting in  $y = 2^u$  values. Therefore, there are  $y - 1$  possible values for which variable  $v_j^i$  can differ from its predecessor  $v_j^{i-1}$ . Therefore, the probability for one of these other values  $x \in d_j^i$  becomes  $p(x) = \frac{1}{k} \times \frac{1}{y-1} = \frac{1}{k(y-1)}$  (this equal probability distribution over the possible values results in higher entropy, but recall that we do not make other assumptions on the nature of the inputs). Of course, there is only one value assignment when the variable  $v_j^i$  does not change, i.e., the valuation of the same variable in the predecessor state  $v_j^{i-1}$ .<sup>2</sup> This results in the following definition of entropy per variable in the stream:

$$H_{var}(v_j^i) = -\frac{k-1}{k} \log_2 \left( \frac{k-1}{k} \right) + \sum_{n=1}^{y-1} -\frac{1}{k(y-1)} \log_2 \left( \frac{1}{k(y-1)} \right)$$

After some simplification, we can derive the state vector’s entropy:

$$H_{state}(v_0^i, \dots, v_{k-1}^i) = \sum_{j=0}^{k-1} H_{var}(v_j^i) = \log_2(y-1) + \log_2(k-1) + k \log_2 \left( \frac{k}{k-1} \right) \tag{1}$$

**Theorem 1 (Information Entropy of States Exhibiting Locality).** For  $k > 1$ , the information entropy of state vectors in state spaces exhibiting locality, abbreviated with  $H_{state}$ , is bound by:

$$\log_2(y-1) + \log_2(k-1) + 1 \leq H_{state} \leq \log_2(y) + \log_2(k) + 2 = u + \log_2(k) + 2$$

<sup>2</sup> The assumption that predecessor is always known of course breaks down for the initial state  $\iota$ . Our model does not account for the initial storage required for  $\iota$ . However, as the number of states  $|V|$  typically grows very large, this space is negligible.

*Proof.* We first show that  $H_{state} \leq \log_2(y) + \log_2(k) + 2 = u + \log_2(k) + 2$ . Simplification using  $\log_2(k - 1) \leq \log_2(k)$  yields:  $(1 + \frac{1}{k-1})^k \leq 4$ . As for  $k = 2$  (recall that  $k > 1$ ), we have  $(1 + \frac{1}{k-1})^k = 4$  and, in the limit, we have  $\lim_{k \rightarrow \infty} (1 + \frac{1}{k-1})^k = \lim_{k \rightarrow \infty} (1 + \frac{1}{k})^k = e$ , it can be seen that  $(1 + \frac{1}{k-1})^k \leq 4$  holds, and hence  $H_{state} \leq \log_2(y) + \log_2(k) + 2 = u + \log_2(k) + 2$ .

Now we show that  $\log_2(y - 1) + \log_2(k - 1) + 1 \leq H_{state}$ .

Simplification yields:  $2 \leq (1 + \frac{1}{k-1})^k$ . Again for  $k = 2$ , we have  $(1 + \frac{1}{k-1})^k = 4$  and  $\lim_{k \rightarrow \infty} (1 + \frac{1}{k-1})^k = e$ , hence  $\log_2(y - 1) + \log_2(k - 1) + 1 \leq H_{state}$ .  $\square$

Intuitively, this approximation makes sense since a single modification in each new state vector can be encoded with solely the index of the changed variable, in  $\log(k)$  bits, plus its new value, in  $\log(y) = u$  bits, plus some overhead to accommodate cases where more than one variable changes value. This result indicates that locality could allow us to store sets of arbitrarily long ( $k \cdot u$ -bit) state vectors using a small integer of less than  $u + \log_2(k) + 2$  bits per vector.

In practice, this could mean that vectors of a thousand (1024) of byte-size variables can be compressed to 20 bits each, which is only slightly more than if these states were numbered consecutively—in which case the states would be 18 bits—but far less than 8192 bits required for storing the full state vectors.

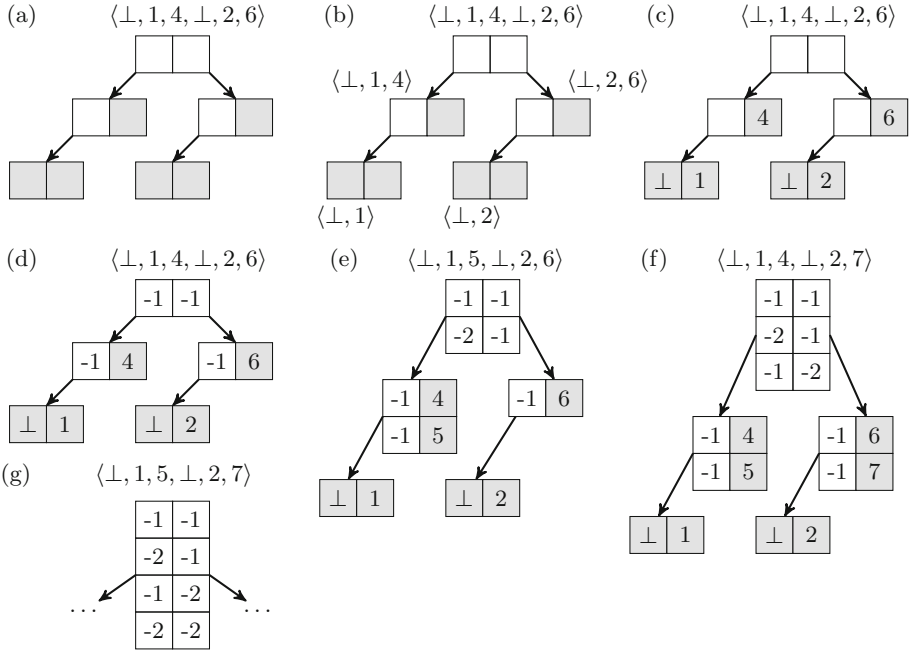
### 3 An Analysis of Binary Tree Compression

The interpretation of the results in Sect. 2 suggests a trivial data structure to reach the information theoretical lower bound: Simply store incremental differences between state vectors. However, as noted in the introduction, an incremental data structure like that does not provide the required efficiency for lookup up operations (the reachability procedure in Algorithm 1 needs to determine whether states have been visited before on Line 7).

The current section shows how many state vectors can be folded into a single binary tree of hash tables to achieve sharing among subvectors, while also achieving poly-logarithmic lookup times in the worst case. This is the first step towards achieving the optimal compression from Sect. 2 in practice. Section 4 presents the second step. We focus here on the analysis of tree compression. For tree algorithms, refer to [21].

The shape of the binary tree is fixed and depends only on  $k$ . Vectors are folded in the tree until only tuples remain. These are stored in the leaves. Using hashing, tuples receive a unique index which is propagated back upwards, forming again new tuple in the tree nodes that can be hashed again. This process continues until a tuple is stored in the root node, representing the entire vector.

Figure 3(a) demonstrates how the state  $\langle \perp, 1, 4, \perp, 2, 6 \rangle$  is folded into an empty tree, which consists of  $k - 1$  nodes of empty hash tables storing tuples. The process starts at the root of the tree (a), and recursively visits children while splitting the vector (b). When the leaves of the tree (colored gray) are reached, they are filled with the values from the vector (c). The vectors inserted into the



**Fig. 3.** Tree folding process for  $\langle \perp, 1, 4, \perp, 2, 6 \rangle$  (in (a)–(d)),  $\langle \perp, 1, 5, \perp, 2, 6 \rangle$  (in (e)),  $\langle \perp, 1, 5, \perp, 2, 7 \rangle$  (in (f)) and  $\langle \perp, 1, 4, \perp, 2, 7 \rangle$  (in (g)).

hash tables can be indexed (we use negative numbers to distinguish indices). Indices are then propagated back upwards to fill the tree until the root (d).

Using a similar process, we can insert vector  $\langle \perp, 1, 5, \perp, 2, 6 \rangle$  (e). The hash tables in the tree nodes extended with index -2 storing  $\begin{bmatrix} -1 & 5 \end{bmatrix}$  in the left child of the root, while the root is extended with the tuple  $\begin{bmatrix} -2 & -1 \end{bmatrix}$ . Notice how sub-vector sharing already occurs since the tuple  $\begin{bmatrix} -1 & 5 \end{bmatrix}$  in the left child of the root points again to  $\begin{bmatrix} \perp & 1 \end{bmatrix}$ . In (f), the vector  $\langle \perp, 1, 4, \perp, 2, 7 \rangle$  is also added. In this case, only the right child of the root needs to be extended, while the tuple  $\begin{bmatrix} -1 & -2 \end{bmatrix}$  is added to the root.

With these three vectors in the tree (f), we can now easily add a new vector  $\langle \perp, 1, 5, \perp, 2, 7 \rangle$  by merely adding the tuple  $\begin{bmatrix} -2 & -2 \end{bmatrix}$  to the root of the tree (g). We observe that an entire state vector (of length  $k$  in general) can be compressed to a single tuple of integers in the root of the tree, provided that the sub-vectors are already present in the left and the right sub-tree of the root.

The tree containing four vectors in Fig. 3 (g) uses 20 “places” (= 10 tuples in tree nodes) to store four vectors with a total of 24 variables. The more vectors are added, the more sharing can occur and the better the compression. We now recall the worst-case and the best-case compression ratio for this tree database. We make the following reasonable assumptions about their dimensions:

- The respective database stores  $n = |V|$  state vectors of  $k$   $u$ -bit variables.
- The size of tree tuples is  $2w$  bits, and  $w$  bits is enough to store both a variable valuation (in a leaf) or a tree reference (in a tree node), hence  $u \leq w$ .
- Keys can be stored without overhead in tables.<sup>3</sup>
- $k$  is a power of 2.<sup>4</sup>

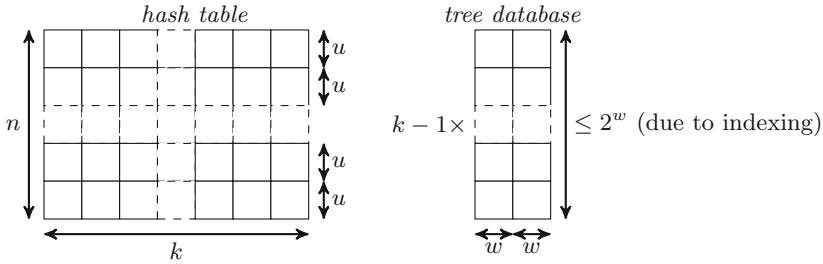


Fig. 4. From left to right: a hash table and a tree table with their dimensions.

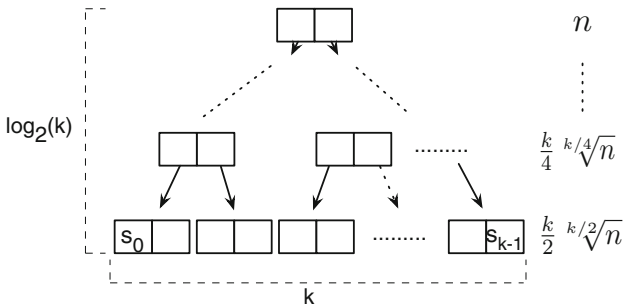


Fig. 5. Optimal entries per tree node level.

Figure 4 provides an overview of the different data structures and the stated assumptions about their dimensions.

To arrive at the worst-case compression scenario (Theorem 2), consider the case where all states  $s \in V$  have  $k$  identical data values:  $V = \{v^k \mid v \in \{1, \dots, n\}\}$ , where  $v^k$  is a vector of length  $k$ :  $\langle v, \dots, v \rangle$ . No sharing can occur between state vectors in the database, so for each state we store  $k - 1$  tuples at the tree nodes.

**Theorem 2** ([4]). *In the worst case, the tree database requires at most  $k - 1$  tuple entries of  $2w$  bits per state vector.*

<sup>3</sup> [21] explains in detail how this can be achieved.

<sup>4</sup> Solely assumed to simplify the formulae below.



**Table 1.** Theoretical bounds for the compressed state sizes in the tree database and in plain hash table storage. Note that while  $u \leq w$ , often  $u, w$  are in the same ballpark.

Structure	Worst case	Best case
Hash table	$ku$	$ku$
Tree database	$2kw - 2w$	$2w + \epsilon w$

The best-case scenario (Theorem 3) is easy to comprehend from the effects of a good combinatorial structure on the size of the parent tables in the tree. If a certain tree table contains  $d$  tuple entries, and its sibling contains  $e$  entries, than the parent can have up to  $d \times e$  entries (all combinations, i.e. the Cartesian product). In a tree that is perfectly balanced ( $d = e$  for all sibling tables), the root node has  $n$  entries (1 per state), its children have  $\sqrt{n}$  entries, its children’s children  $\sqrt[4]{n}$ , etc. Figure 5 depicts this scenario.

Hence there are a total of  $n + 2\sqrt{n} + 4\sqrt[4]{n} + \dots + (\log_2(k)\text{times}) \dots + k/2 \sqrt[k/2]{n}$  tuple entries. Dividing this series by  $n$  gives a series for the expected number of tuple entries per state:  $\sum_{i=0}^{\log_2(k)-1} 2^i \frac{\sqrt[2^i]{n}}{n}$ . It is hard to see where this series exactly converges, but Theorem 3 provides an upper bound. The theorem is a refinement of the upper bound established in [4]. Note that the example above of a tree with the four Bakery algorithm states already represents an optimal scenario, i.e., the root table is the cross product of its children.

**Theorem 3.** *In the best case and with  $k \geq 8$ , the tree database requires less than  $n + 2\sqrt{n} + 2\sqrt[4]{n}(k - 4)$  tuple entries of  $2w$  bits to store  $n$  vectors.*

*Proof.* In the best case, the root tree table contains  $n$  entries and its children both contain  $\sqrt{n}$  entries. The entries in the 4 children’s children of the root represent vectors of size  $k/4$ . These 4 tree nodes contain each  $\sqrt[4]{n}$  entries that each require  $k/4 - 1$  tuples taking the worst case according to Theorem 2 (hence also  $k \geq 8$ ). □

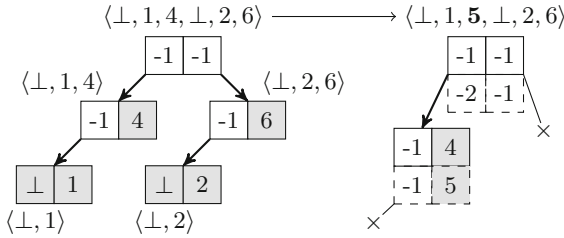
**Corollary 1** ([21]). *In the best case, the total number of tuple entries  $l$  in all descendants of root table is negligible ( $l \ll n$ ), assuming a relatively large number of vectors is stored:  $n \gg k^2 \gg 1$ .*

**Corollary 2** ([21]). *In the best case, the compressed state size approaches  $2w$ .*

Table 1 lists the achieved compressed sizes for states, as stored in a normal hash table and a tree database. As a simplifying assumption, we take  $u$  to be equal  $w$ , which can be the case if the tree is specifically adapted to accommodate  $u$  bit references.

**Performance.** We conclude the current section with a note on the performance of the tree database compared to a plain hash table. The tree trades  $ku$  bit vector lookups for  $k - 1$  of  $2u$ -bit tuple lookups in its nodes. Apart from the additional

data access required  $(ku - 2u)$ , it seems like the increased random memory accesses could cause poor behavior on modern CPUs. However, in the case of good compressions, the lower tables in the tree typically contain fewer entries which can more easily be cached, whereas effective caching of the large plain vectors in hash table solutions is nigh impossible. Moreover, we can further use locality to speed up tree lookups by keeping the tree of the predecessor state in the search stack ( $Q$ ), as explained in [21]. Figure 6 illustrates this.



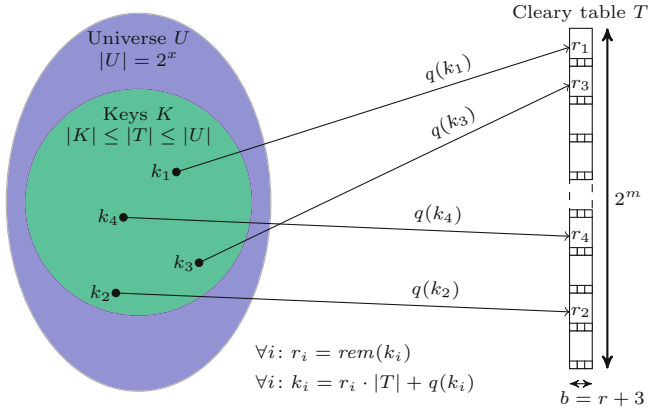
**Fig. 6.** Incremental insertion of state  $\langle \perp, 1, 5, \perp, 2, 6 \rangle$ . Only a small part of the tree needs to be updated (dashed boxes), because the predecessor state  $\langle \perp, 1, 4, \perp, 2, 6 \rangle$  is used to lookup unchanged parts (the crosses in the tree of the successor state).

### 4 A Novel Compact Tree

The current section shows how a normal tree database can be extended to reach the information theoretical optimum using a compact hash table.

**Hash Tables and Compact Hash Tables.** A hash table stores a subset of a large *universe*  $U$  of keys and provides the means to lookup individual keys in constant time. It uses a *hash function* to calculate an address  $h$  from the unique *key*. The entire key is then stored at its hash or home location in a table  $T$  (an array of *buckets*):  $T[h] \leftarrow key$ . Because typically  $|U| \gg |T|$ , multiple keys may have the same hash location. These so-called *collisions* are handled by calculating alternate hash locations and inserting the key there if empty. This process is known as *probing*. For this reason, the entire key needs to be stored in it; to distinguish which key is currently mapped to a bucket of  $T$ .

Observe, however, that in the case that  $|U| \leq |T|$ , the table can be replaced with a *perfect hash function* and a bit array. *Compact hashing* [8] generalizes this idea for the case  $|U| > |T|$  (the table size is relatively close to the size of the universe). The compact table first splits a key  $k$  into a quotient  $q(k)$  and a remainder  $rem(k)$ , using a reversible operation, e.g.,  $q(k) = k \% |T|$  and  $rem(k) = k / |T|$ . When the key is  $x = \lceil \log_2(|U|) \rceil$  bits, the quotient  $m = \lceil \log_2(|T|) \rceil$  bits and the remainder  $r = x - m$  bits. The quotient is used for addressing in  $T$  (like in a normal hash table). Now only the remainder is stored in the bucket. The complete key can now be reconstructed from the value in  $T$  and the home location of the key.



**Fig. 7.** Cleary table  $T$  storing keys  $K$  from universe  $U$  with three admin. bits/bucket. (We omit that keys should be hashed, with invertible function, for good distribution.)

If, due to collisions, the key is not stored at its home location, additional information is needed. Cleary [8] solved this problem with little overhead by imposing an order on the keys in  $T$  and introducing three administration bits per bucket. For details, see [8, 12, 28]. Because of the administration bits, the bucket size  $b$  of compact hash tables is  $b = r + 3$  bits. The ratio  $b/x$  can approach zero arbitrarily close, yielding good compression. For instance, a compact table only needs 5 bits per bucket to store  $2^{30}$  32-bit keys (Fig. 7).

**Compact Tree Database.** To create a compact tree database, we replace the hash tables in the tree nodes with compact hash tables.

Let the tree references again be  $w$  bits; Tuples in a tree node table are  $2w$  bits. The tree node table’s universe therefore contains  $2^{2w}$  tuples. However, tree node tables cannot contain more than  $2^w$  entries, otherwise the entries cannot be referenced (with  $w$ -bit indices) by parent tree node tables. As the tree’s root table has no parent, it can contain up to  $2^{2w}$  entries. Let  $o$  be the overcommit of the tree root table  $T_{root}$ , i.e.,  $\log_2(|T_{root}|) = 2^{w+o}$  for  $0 \leq o \leq w$ .

Overcommitting the root table in the tree can yield better reductions as we will see. However, it also limits the subsets of the state universe that the tree can store. Close-to-worst-case subsets might be rejected as the left or right child ( $2^w$  tuples max) of the root can grow full before the root does ( $2^{w+o}$  tuples max).

We will only focus on replacing the root table with a compact hash table as it dominates the tree’s memory usage in the optimal case according to Corollary 1. The following parameters follow for using a compact hash tables for  $T_{root}$ :

- $x = 2w$ , (universe bits)
- $m = w + o$ , (quotient bits)
- $r = 2w - w - o = w - o$ , and (remainder bits)
- $b = 2w - w - o + 3 = w - o + 3$ . (bucket bits)

Let the Compact Tree Database be a Tree Database with the root table replaced by a compact hash table with the dimensions provided above, ergo:  $n = |V| = |T_{root}| = 2^{w+o} = 2^m$ . Theorem 4 gives its best-case memory usage.

**Theorem 4 (Compact Tree Best-Case).** *In the best case and with  $k \geq 8$ , the compact tree database requires less than  $CT_{opt} \triangleq (w - o + 3)n + 4w\sqrt{n} + 4w\sqrt[4]{n}(k - 4)$  bits to store  $n$  vectors.*

*Proof.* According to Theorem 3, there are at most  $n + 2\sqrt{n} + 2\sqrt[4]{n}(k - 4)$  tuples in a tree with optimal storage. The root table contains  $n$  of these tuples, its descendants use at most  $2\sqrt{n} + 2\sqrt[4]{n}(k - 4)$  bits. The  $n$  tuples in the root table can now be stored using  $w - o + 3$  bits in the compact hash table buckets instead of  $2w$  bits, hence the root table uses  $n(w - o + 3)$  bits.  $\square$

Finally, Theorem 5 relates the compact tree compression results to our information theoretical model in Sect. 2, under the reasonable assumptions that  $8 \leq k \leq \sqrt[4]{n} + 4$ . As a consequence, when the overcommit ( $o - 7$  bits) fills the gap of  $w - u$  bits between the sizes of references in the tree ( $w$  bits) and the sizes of variables ( $u$  bits), the optimal compression of the compact tree is approached. If  $o - 7 > w - u$ , the compact tree can even surpass the compression predicted by our information theoretical model. This is not surprising as the tree with  $k = 2$  reduces to a compact hash table, for which a different information theoretical model holds [12, 27].

**Theorem 5.** *Let  $CT_{opt}$  be the best-case compact-tree compressed vector sizes. We have  $CT_{opt} \leq \frac{w-o+7}{u}H_{state}$  provided  $8 \leq k \leq \sqrt[4]{n} + 4$ .*

*Proof.* According to Theorem 1,  $nH_{state} \leq un + \log_2(k)n + 2n$  bits. According to Theorem 4, the compact tree database uses at most  $CT_{opt} \triangleq (w - o + 3)n + 4w\sqrt{n} + 4w\sqrt[4]{n}(k - 4)$  bits in the best case and with  $k \geq 8$ .

We show that  $CT_{opt} \leq cH_{state}$  using lower bound Theorem 1 and derive  $c$ .

After simplification using  $(u - 1) \leq \log_2(y - 1)$  and  $0 \leq \log_2(k - 1)$ , we obtain:  $4w/\sqrt{n} + 4w(k - 4)/n^{3/4} \leq cu - w + o - 3$ . As the premise ensures that  $n \geq (k - 4)^4$ , this can be further simplified to  $4w/\sqrt{n} + 4w\sqrt[4]{n}/n^{3/4} \leq cu - w + o - 3$  and then to  $8w/\sqrt{n} \leq cu - w + o - 3$ .

In an intermediate step, we show that  $w/\sqrt{n} \leq 1/2$  under the premises  $n \geq (k - 4)^4$  and  $k \geq 8$ . We have  $w \leq \log_2(n)$  in order to accommodate the worst-case compression (see Theorem 2 in Sect. 4). Therefore, we can also prove  $\log_2(n)/\sqrt{n} \leq 1/2$ . Implied by the two earlier assumptions from the premise, we have  $n \geq 256$  for which the inequality indeed holds.

With  $w/\sqrt{n} \leq 1/2$ , the above gives  $4 \leq cu - w + o - 3$  and  $\frac{w-o+7}{u} \leq c$ .

Therefore, we obtain  $CT_{opt} \leq \frac{w-o+7}{u}H_{state}$ , provided that  $n \geq (k - 4)^4$ .  $\square$

## 5 Experiments

We implemented the Compact Tree in the model checker LTSMIN [22]. This implementation is based on two concurrent data structures: a tree database [21]

and a compact hash table [28], based on Cleary’s approach [8]. The parameters of the Compact Tree Table in this implementation are (for details see [23]):

- $w = 30$  bits (The internal tree references are 30 bit)
- $u = 30$  bits (The state variables can be 30-bit integers, often less is used)
- $o = 2$  bits (The root table fits a maximum of  $2^{32}$  elements)

LTSMIN is a language-independent model checker based on a *partitioned next-state interface* [18]. We exploit this property to investigate the compression ratios of the Compact Tree for four different input types: DVE models written for the DIVINE model checker [1], Promela models written for the SPIN model checker [15], process algebra models written for the MCRL2 model checker [9], and Petri net models from the MCC contest [20]. Table 2 provides an overview of the models in each of these input formats and a justification for the selection criterion used. In total, over 400 models were used in these benchmarks.

**Table 2.** Input languages and model selection criteria

DVE	All 267 benchmarks from the BEEM database [24] that completed within one hour in (sequential) LTSMIN are selected. (This selection criterium is more stringent than for the other languages, because the set of models is large and the presence of differently sized versions of the same type of model still ensures that the selection is varied.)
Promela	All models currently supported by LTSMIN [2] with the same state count as in SPIN are selected. This includes case studies of the GARP, the i-, x509 and BRP protocols
Petri net	All models from the MCC 2016 competition [20] that are also considered by Jensen et al. [16] and complete within 10 h in (sequential) LTSMIN. (Again this ensures a varied selection, since Jensen et al. [16] only feature instances that resulted in best-case, worst-case and average-case compression using a Trie data structure.)
MCRL2	We selected all industrial case studies from the MCRL2 toolset that completed within 10 h in (sequential) LTSMIN

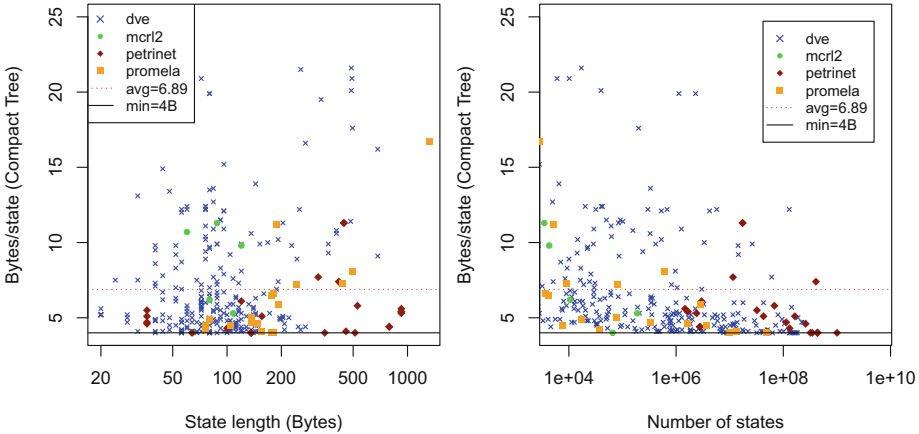
All experiments ran on a machine with 128 GB memory and 48 cores: four AMD Opteron™ 6168 processors with 12 cores each.

**Compression Ratio.** Compressed state sizes of our implementation can roughly approach  $w - 2 + 3 = 31$  bits or  $\pm 4$  Bytes by Corollary 1 and Theorem 4. We first investigate whether this compression is actually reached in practice. Figure 8 plots the compressed sizes of the state vectors against the length of the uncompressed vector. We see that for some models the optimal compression is indeed reached. The average compression is 7.88 Bytes per state. The fact that there is little correlation with the vector length confirms that the compressed size indeed tends to be constant and vectors of up to 1000 Bytes are compressed

to just above 4 Bytes. Figure 9 furthermore reveals that good compression correlates positively with the state space size, which can be expected as the tree can exhibit more sharing.

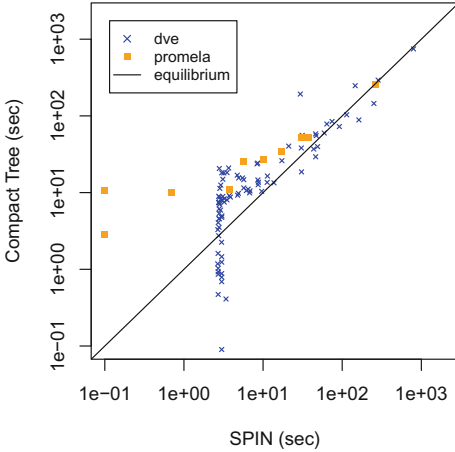
Only for Petri nets and for DVE models, we find models that exhibit worse compression (between 10 and 15 Bytes per state) even when the state space is large. However, we observed that in these cases, the vector length  $k$  is also large, e.g., the two Petri net instances with a compressed size of around 12 have  $k > 400$ . Based on some earlier informal experiments, we believe that with some variable reordering, these compression might very well be improved to reach the optimum. Thus far, however, we were unable to derive a reordering heuristic that consistently improves the compression.

**Runtime Performance and Parallel Scalability.** In the introduction, we mentioned the requirement that a database visited set ideally features constant lookup times, like in a normal hash table. To this end, we compare the runtime of the DVE models with the SPIN model checker; a model checker known for its fast state generator.<sup>5</sup> Figure 10 confirms that the runtimes of LTSMIN with Compact Tree are sequentially on par with those of SPIN, and often even better. We attribute this performance mainly to the incremental vector insertion discussed in Sect. 3 (see Fig. 6). Based on the MCC 2016 [20] results, we believe that LTSMIN’s performance is on par with other Petri net tools as well.

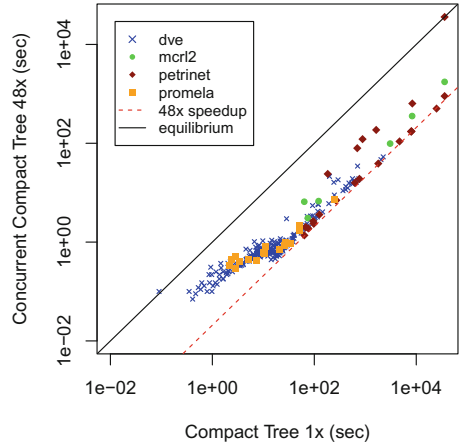


**Fig. 8.** Compressed sizes in Compact Tree for all benchmarks against the length  $k$  of the uncompressed state vector. **Fig. 9.** Compressed sizes in Compact Tree for all benchmarks against the size  $n$  of state space.

<sup>5</sup> The DVE models are translated to Promela and we only selected those (76/267) which preserved state count. This comparison can be examined interactively at <http://fmt.ewi.utwente.nl/tools/ltsmin/performance/> (select LTSmin-clearly-dfs).



**Fig. 10.** Sequential runtimes of LTSMIN with Compact Tree and SPIN with optimal settings (as reported in [2]) on (translated) DVE models and Promela models.

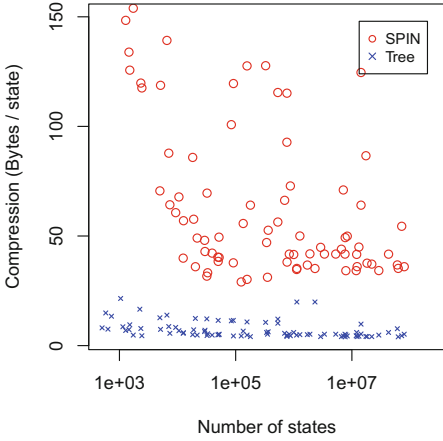


**Fig. 11.** Runtimes, sequentially and with 48 threads, of LTSMIN with compact tree on all models: DVE, Promela, process algebra and Petri nets.

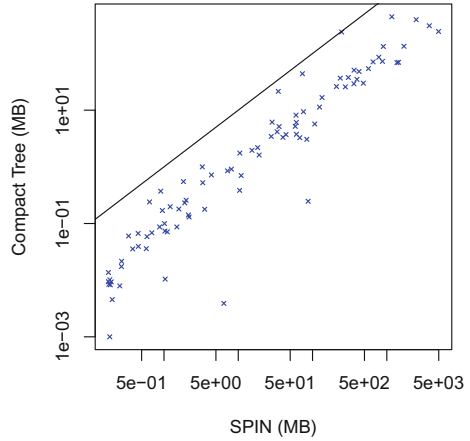
The measured performance first of all confirms that the Compact Tree satisfies its requirements. Secondly, it provides a good basis for the analysis of parallel scalability (if we had chosen to implement the Compact Tree in a slow scripting language, the slowdown would yield “free” speedup). Figure 11 compares the sequential runtimes to the runtimes with 48 threads. The measured speedup often surpasses 40x, especially when the runtimes are longer and there is more work to parallelize. Speedups are good regardless of input language.

**Comparison with Other Data Structures.** SPIN’s collapse compression uses the structure in the model to fold vectors, similar as in tree compression. The lower bounds reported in the current paper cannot be reached with collapse due to the  $n$ -ary tree and the two levels. Figures 12 and 13 show additional experiments that show an order of magnitude difference in practice.

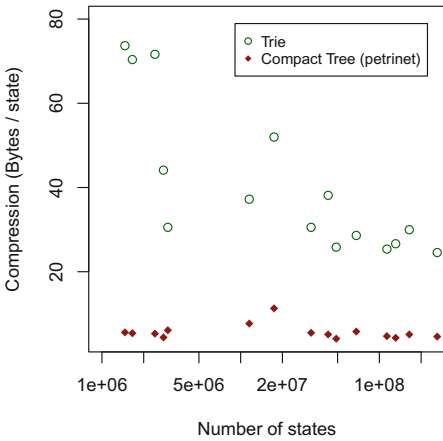
Jensen et al. [16] propose a Trie for storing state vectors. Tries compress vectors by ensuring sharing between prefixes. BDDs [6] also store state vectors efficiently, however, Jensen et al. [17] figure them too slow for state space exploration. We compared both Tries and BDDs with the Compact Tree and found that (1) the Trie’s compression is less than the Compact Tree though sometimes faster (Figs. 14 and 15), and (2) that BDD’s are not *always* prohibitively expensive with LTSMIN (because it learns the transition relation [18]), but nonetheless hard to compare to Tree Compression (Figs. 17 and 16).



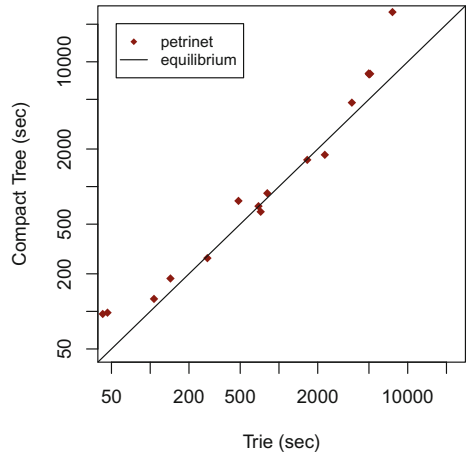
**Fig. 12.** Compressed sizes per state of LTSMIN with Compact Tree and SPIN with collapse compression [14] on DVE models.



**Fig. 13.** Absolute memory use of LTSMIN with Compact Tree and SPIN with collapse compression [14] on DVE models.

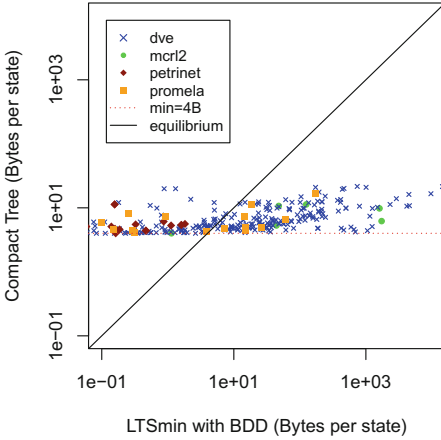


**Fig. 14.** Memory use per state of LTSMIN with Compact Tree and Trie from [16] on Petri net models.

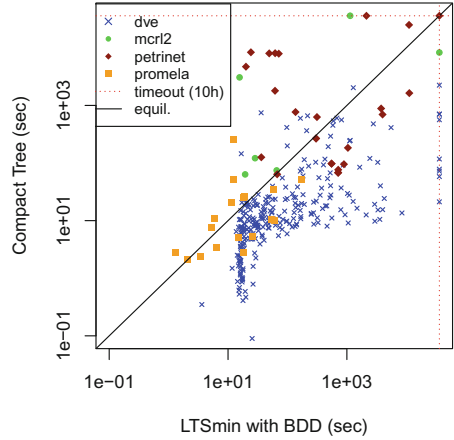


**Fig. 15.** Runtime (sequential) of LTSMIN with Compact Tree and Trie from [16] on Petri net models.





**Fig. 16.** Memory use per state of LTSMIN with Compact Tree and BDD [3] on MCRL2, Promela and Petri net models.



**Fig. 17.** Runtime (sequential) of LTSMIN with Compact Tree and BDD [3] on MCRL2, Promela and Petri net models.

## 6 Discussion and Conclusion

The tree compression method discussed here is a more general variant of recursive indexing [14], which only breaks down processes into separate tables. Hash compaction [25] compresses states to an integer-sized hash, but this lossy technique becomes redundant with the compact tree database. Bloom filters [13] still present a worthwhile lossy alternative using only a few bits per state, but of course abandon soundness when applied in model checking.

Evangelista et al. [11] report on a hash table storing incremental differences of successor states (similar to the incremental data structure discussed in Sect. 3). Their partial vectors take  $2u + \log(E)$  bits, where  $E$  is the set of (deterministic) actions in the model. Defying our requirement of poly-logarithmic for lookups, Evangelista et al. reconstruct full states by reconstructing all ancestors.

A Binary Decision Diagram (BDD) [6] can store an astronomically sized state set using only constant memory (the true leaf). Our information theoretical model suggests however that compressed sizes are merely linear in the number of states (and constant in the length of the state vector). We can explain this with the fact that we only assume locality about inputs. Compression in BDDs, on the other hand, depends on the entire state space. Therefore, we would have to assume structural, global properties to describe the non-linear compression of BDDs (e.g. the input's decomposition into processes, symmetries, etc.).

Much like in BDDs [5], the variable ordering influences the number of nodes in a tree table and thus the compression, as mentioned in Sect. 1. Consider the vector set  $\{i, i, j, j \mid i, j \in [1 \dots N]\}$ : Only the root node in a compact tree will contain  $N^2$  entries, while the leaf nodes contain  $N$  entries. On the other hand, we have no such luck for the set  $\{i, j, i, j \mid i, j \in [1 \dots N]\}$ . Preliminary research [29]

revealed that the tree's optimum can be reached in most cases for DVE models, but we were unable to find a heuristic to consistently realize this.

**Acknowledgements.** The author thanks Yakir Vizel for promptly pointing out the natural number as a limit and Tim van Erven for a fruitful discussion.

## References

1. Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkal, P., Štill, V.: Model checking of C and C++ with DIVINE 4. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 201–207. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-68167-2\\_14](https://doi.org/10.1007/978-3-319-68167-2_14)
2. van der Berg, F., Laarman, A.: SpinS: extending LTSmin with Promela through SpinJa. ENTCS **296**, 95–105 (2013)
3. 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](https://doi.org/10.1007/978-3-642-14295-6_31)
4. Blom, S., Lisser, B., van de Pol, J., Weber, M.: A database approach to distributed state space generation. ENTCS **198**(1), 17–32 (2008)
5. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Trans. Comput. **45**, 993–1002 (1996)
6. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
7. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking:  $10^{20}$  states and beyond. In: LICS, pp. 428–439 (1990)
8. Cleary, J.G.: Compact hash tables using bidirectional linear probing. IEEE Trans. Comput. **C-33**(9), 828–834 (1984)
9. Cranen, S., Groote, J.F., Keiren, J.J.A., Stappers, F.P.M., de Vink, E.P., Weselink, W., Willemse, T.A.C.: 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](https://doi.org/10.1007/978-3-642-36742-7_15)
10. Emerson, E.A., Wahl, T.: Dynamic symmetry reduction. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 382–396. Springer, Heidelberg (2005). [https://doi.org/10.1007/978-3-540-31980-1\\_25](https://doi.org/10.1007/978-3-540-31980-1_25)
11. Evangelista, S., Kristensen, L.M., Petrucci, L.: Multi-threaded explicit state space exploration with state reconstruction. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 208–223. Springer, Cham (2013). [https://doi.org/10.1007/978-3-319-02444-8\\_16](https://doi.org/10.1007/978-3-319-02444-8_16)
12. Geldenhuys, J., Valmari, A.: A nearly memory-optimal data structure for sets and mappings. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 136–150. Springer, Heidelberg (2003). [https://doi.org/10.1007/3-540-44829-2\\_9](https://doi.org/10.1007/3-540-44829-2_9)
13. Holzmann, G.J.: An analysis of bitstate hashing. In: Dembiński, P., Średniawa, M. (eds.) PSTV 1995. IFIPAICT, pp. 301–314. Springer, Boston (1996). [https://doi.org/10.1007/978-0-387-34892-6\\_19](https://doi.org/10.1007/978-0-387-34892-6_19)
14. Holzmann, G.J.: State compression in SPIN: recursive indexing and compression training runs. In: Proceedings of 3rd International SPIN Workshop (1997)
15. Holzmann, G.J.: The model checker SPIN. IEEE TSE **23**, 279–295 (1997)

16. Jensen, P.G., Larsen, K.G., Srba, J.: PTrie: data structure for compressing and storing sets via prefix sharing. In: Hung, D., Kapur, D. (eds.) ICTAC 2017. LNCS, vol. 10580, pp. 248–265. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-67729-3\\_15](https://doi.org/10.1007/978-3-319-67729-3_15)
17. Jensen, P.G., Larsen, K.G., Srba, J., Sørensen, M.G., Taankvist, J.H.: Memory efficient data structures for explicit verification of timed systems. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 307–312. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-06200-6\\_26](https://doi.org/10.1007/978-3-319-06200-6_26)
18. 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](https://doi.org/10.1007/978-3-662-46681-0_61)
19. Katz, S., Peled, D.: An efficient verification method for parallel and distributed programs. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1988. LNCS, vol. 354, pp. 489–507. Springer, Heidelberg (1989). <https://doi.org/10.1007/BFb0013032>
20. Kordon, F., et al.: Complete results for the 2016 edition of the model checking contest, June 2016. <http://mcc.lip6.fr/2016/results.php>
21. Laarman, A., van de Pol, J., Weber, M.: Parallel recursive state compression for free. In: Groce, A., Musuvathi, M. (eds.) SPIN 2011. LNCS, vol. 6823, pp. 38–56. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-22306-8\\_4](https://doi.org/10.1007/978-3-642-22306-8_4)
22. Laarman, A., van de Pol, J., Weber, M.: Multi-core LTSMIN: marrying modularity and scalability. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 506–511. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-20398-5\\_40](https://doi.org/10.1007/978-3-642-20398-5_40)
23. Laarman, A.: Scalable multi-core model checking. Ph.D. thesis, UTwente (2014)
24. Pelánek, R.: BEEM: benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-73370-6\\_17](https://doi.org/10.1007/978-3-540-73370-6_17)
25. Stern, U., Dill, D.L.: Improved probabilistic verification by hash compaction. In: Camurati, P.E., Evesking, H. (eds.) CHARME 1995. LNCS, vol. 987, pp. 206–224. Springer, Heidelberg (1995). [https://doi.org/10.1007/3-540-60385-9\\_13](https://doi.org/10.1007/3-540-60385-9_13)
26. Valmari, A.: Error detection by reduced reachability graph generation. In: APN, pp. 95–112 (1988)
27. Valmari, A.: What the small Rubik’s cube taught me about data structures, information theory, and randomisation. STTT **8**(3), 180–194 (2006)
28. van der Vegt, S., Laarman, A.: A parallel compact hash table. In: Kotásek, Z., Bouda, J., Černá, I., Sekanina, L., Vojnar, T., Antoš, D. (eds.) MEMICS 2011. LNCS, vol. 7119, pp. 191–204. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-25929-6\\_18](https://doi.org/10.1007/978-3-642-25929-6_18)
29. de Vries, S.H.S.: Optimizing state vector compression for program verification by reordering program variables. In: 21st Twente SConIT, vol. 21, 23 June 2014
30. Wahl, T., Donaldson, A.: Replication and abstraction: symmetry in automated formal verification. Symmetry **2**(2), 799–847 (2010)



# Stubborn Transaction Reduction

Alfons Laarman<sup>(✉)</sup> 

Leiden University, Leiden, The Netherlands

a.w.laarman@liacs.leidenuniv.nl

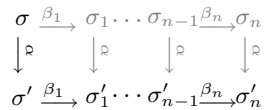
**Abstract.** The exponential explosion of parallel interleavings remains a fundamental challenge to model checking of concurrent programs. Both partial-order reduction (POR) and transaction reduction (TR) decrease the number of interleavings in a concurrent system. Unlike POR, transactions also reduce the number of intermediate states. Modern POR techniques, on the other hand, offer more dynamic ways of identifying commutative behavior, a crucial task for obtaining good reductions.

We show that transaction reduction can use the same dynamic commutativity as found in stubborn set POR. We also compare reductions obtained by POR and TR, demonstrating with several examples that these techniques complement each other. With an implementation of the dynamic transactions in the model checker LTSmin, we compare its effectiveness with the original static TR and two POR approaches. Several inputs, including realistic case studies, demonstrate that the new dynamic TR can surpass POR in practice.

## 1 Introduction

POR [20,33,46] yields state space reductions by selecting a subset  $P_\sigma$  of the enabled actions  $E_\sigma$  at each state  $\sigma$ ; the other enabled actions  $E_\sigma \setminus P_\sigma$  are pruned. For instance, reductions preserving deadlocks (states without outgoing transitions) can be obtained by ensuring the following properties for the set  $P_\sigma \subseteq E_\sigma \subseteq A$ , where  $A$  is the set of all actions:

- In any state  $\sigma_n$  reachable from  $\sigma$  via pruned actions  $\beta_1, \dots, \beta_n \in A \setminus P_\sigma$ , all actions  $\alpha \in P_\sigma$  commute with the pruned actions  $\beta_1, \dots, \beta_n$  and
- at least one action  $\alpha \in P_\sigma$  remains enabled in  $\sigma_n$ .



The first property ensures that the pruned actions  $\beta_1, \dots, \beta_n$  are still enabled after  $\alpha$  and lead to the same state ( $\sigma'_n$ ), i.e., the order of executing  $\beta_1, \dots, \beta_n$  and  $\alpha$  is irrelevant. The second avoids that deadlocks are missed when pruning

This work is partially supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grant VRG11-005.

This work is part of the research programme VENI with project number 639.021.649, which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).



states  $\sigma_1, \dots, \sigma_n$ . To compute the POR set  $P_\sigma$  without computing pruned states  $\sigma_1, \dots, \sigma_n$  (which would defeat the purpose of the reduction it is trying to attain in the first place), *Stubborn POR uses static analysis to ‘predict’ the future from  $\sigma$ , i.e., to over-estimate the  $\sigma$ -reachable actions  $A \setminus P_\sigma$ , e.g.:  $\beta_1, \dots, \beta_n$ .*

Lipton or transaction reduction (TR) [38], on the other hand, identifies sequential blocks in the actions  $A_i$  of each thread  $i$  that can be grouped into transactions. A transaction  $\alpha_1.. \alpha_k.. \alpha_n \in A_i^*$  is replaced with an atomic action  $\alpha$  which is its sequential composition, i.e.  $\alpha = \alpha_1 \circ .. \circ \alpha_k \circ .. \circ \alpha_n$ . Consequently, any trace  $\sigma_1 \xrightarrow{\alpha_1} \sigma_2 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_k} \dots \xrightarrow{\alpha_{n-1}} \sigma_n \xrightarrow{\alpha_n} \sigma_{n+1}$  is replaced by  $\sigma_1 \xrightarrow{\alpha} \sigma_{n+1}$ , making state  $\sigma_2, \dots, \sigma_n$  *internal*. Thereby, internal states disallow all interleavings of other threads  $j \neq i$ , i.e., *remote actions  $A_j$  are not fired at these states*. Similar to POR, this pruning can reduce reachable states. Additionally, internal states can also be discarded when irrelevant to the model checking problem.

In the database terminology of origin [40], a transaction must consist of:

- A *pre-phase*, containing actions  $\alpha_1.. \alpha_{k-1}$  that may gather required resources,
- a single *commit action*  $\alpha_k$  possibly interfering with remote actions, and
- a *post-phase*  $\alpha_{k+1}.. \alpha_n$ , possibly releasing resources (e.g. via unlocking them).

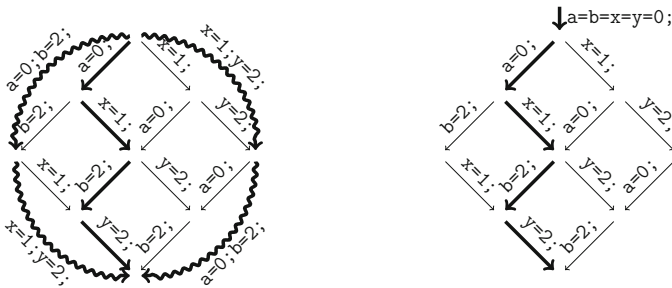
In the pre- and post-phase, the actions (of a thread  $i$ ) must commute with all remote behavior, i.e. *all actions  $A_j$  of all other threads  $j \neq i$  in the system*.

*TR does not dynamically ‘predict’ the possible future remote actions, like POR does.* This makes the commutativity requirement needlessly stringent, as the following example shows: Consider *program1* consisting of two threads. All actions of one thread commute with all actions of the other because only local variables are accessed. Figure 1 (left) shows the POR and TR of this system.

```

program1 := if (fork()) { a = 0; b = 2; } else { x = 1; y = 2; }
program2 := a = b = x = y = 0; if (fork()) { program1; }
    
```

Now assume that a parallel assignment is added as initialization code yielding *program2* above. Figure 1 (right) shows again the reductions. Suddenly, all actions of both threads become dependent on the initialization, i.e. neither action



**Fig. 1.** Transition systems of *program1* (left) and *program2* (right). Thick lines show optimal (Stubborn set) POR. Curly lines show a TR (not drawn in the right figure).

$\mathbf{a} = 0$ ; nor action  $\mathbf{b} = 2$ ; commute with actions of other threads, spoiling the formation of a transaction  $\text{atomic}\{\mathbf{a} = 0; \mathbf{b} = 2;\}$  (idem for  $\text{atomic}\{\mathbf{x} = 1; \mathbf{y} = 2;\}$ ). Therefore, TR does not yield any reduction anymore (not drawn). Stubborn set POR [45], however, still reduces *program2* like *program1*, because, using static analysis, it ‘sees’ that the initialization cannot be fired again.<sup>1</sup>

In the current paper, we show how TR can be made dynamic in the same sense as stubborn set POR [46], so that the previous example again yields the maximal reduction. Our work is based on the prequel [25], where we instrument programs in order to obtain dynamic TR for *symbolic model checking*. While [25] premiered dynamically growing and shrinking transactions, its focus on symbolic model checking complicates a direct comparison with other dynamic techniques such as POR. The current paper therefore extends this technique to enumerative model checking, which allows us to get rid of the heuristic conditions from [25] by replacing them with the more general stubborn set POR method. While we can reduce the results in the current paper to the reduction theorem of [25], the new focus on enumerative model checking provides opportunities to tailor reductions on a per-state basis and investigate TR more thoroughly.<sup>2</sup> This leads to various contributions:

1. A ‘Stubborn’ TR algorithm (STR) more dynamic/general than TR in [25].
2. An open source implementation of (stubborn) TR in the model checker LTSMIN.
3. Experiments comparing TR and POR for the first time in Sect. 5.

Moreover, in Sect. 4, we show analytically that unlike stubborn POR:

1. Computing optimal stubborn TR is tractable and reduction is not heuristic.
2. Stubborn TR can exploit right-commutativity and prune (irrelevant) deadlocks (while still preserving invariants as per Theorem 4).

On the other hand, stubborn POR is still more effective for checking for absence of deadlocks and reducing massively parallel systems. Various open problems, including the combination of TR and POR, leave room for improvement.

Proofs of theorems can be found in the eponymous technical report on Arxiv.

## 2 Preliminaries

**Concurrent transition systems.** We assume a general process-based semantic model that can accommodate various languages. A concurrent transition system (CTS) for a finite set of processes  $P$  is tuple  $\text{TS} \triangleq \langle S, T, A, \sigma_0 \rangle$  with finitely many actions  $A \triangleq \bigsqcup_{i \in P} A_i$ . Transitions are relations between states and actions:

<sup>1</sup> *program2* is a simple example. Yet various programming patterns lead to similar behavior, e.g.: lazy initialization, atomic data structure updates and load balancing [25].

<sup>2</sup> Symbolic approaches can be viewed as reasoning over sets of states, and therefore cannot easily support fine-grained per-state POR/TR analyses.

$T \subseteq S \times A \times S$ . We write  $\alpha_i$  for  $\alpha \in A_i$ ,  $\sigma \xrightarrow{\alpha}_i \sigma'$  for  $\langle \sigma, \alpha_i, \sigma' \rangle \in T$ ,  $T_i$  for  $T \cap (S \times A_i \times S)$ ,  $T_\alpha$  for  $T \cap (S \times \{\alpha\} \times S)$ ,  $\xrightarrow{\alpha}$  for  $\{\langle \sigma, \sigma' \rangle \mid \langle \sigma, \alpha, \sigma' \rangle \in T\}$ , and  $\xrightarrow{i}$  for  $\{\langle \sigma, \sigma' \rangle \mid \langle \sigma, \alpha, \sigma' \rangle \in T_i\}$ .

State space exploration can be used to show invariance of a property  $\varphi$ , e.g., expressing mutual exclusion, written:  $\mathcal{R}(\text{TS}) \models \varphi$ . This is done by finding all reachable states  $\sigma$ , i.e.,  $\mathcal{R}(\text{TS}) \triangleq \{\sigma \mid \sigma_0 \xrightarrow{*} \sigma\}$ , and show that  $\sigma \in \varphi$ .

**Notation.** We let  $en(\sigma)$  be the set of actions enabled at  $\sigma$ :  $\{\alpha \mid \exists \langle \sigma, \alpha, \sigma' \rangle \in T\}$  and  $\overline{en}(\sigma) \triangleq A \setminus en(\sigma)$ . We let  $R \circ Q$  and  $RQ$  denote the *sequential composition* of two binary relations  $R$  and  $Q$ , defined as:  $\{(x, z) \mid \exists y: (x, y) \in R \wedge (y, z) \in Q\}$ . Let  $R \subseteq S \times S$  and  $X \subseteq S$ . Then left restriction of  $R$  to  $X$  is  $X \parallel R \triangleq R \cap (X \times S)$  and right restriction is  $R \parallel X \triangleq R \cap (S \times X)$ . The complement of  $X$  is denoted  $\overline{X} \triangleq S \setminus X$  (the universe of all states remains implicit in this notation). The inverse of  $R$  is  $R^{-1} \triangleq \{\langle x, y \rangle \mid \langle y, x \rangle \in R\}$ .

**POR relations.** Dependence is a well-known relation used in POR. Two actions  $\alpha_1, \alpha_2$  are dependent if there is a state where they do not commute, hence we first define commutativity. Let  $c \triangleq \{\sigma \mid \exists \langle \sigma, \alpha_1, \sigma' \rangle, \langle \sigma, \alpha_2, \sigma'' \rangle \in T\}$ . Now:

$$\begin{aligned}
 \alpha_1 \xrightarrow{\leftrightarrow} \alpha_2 &\triangleq c \parallel \alpha_1 \circ \alpha_2 = c \parallel \alpha_2 \circ \alpha_1 && (\alpha_1, \alpha_2 \text{ strongly-commute}) \\
 \alpha_1 \xrightarrow{\bowtie} \alpha_2 &\triangleq \alpha_1 \circ \alpha_2 = \alpha_2 \circ \alpha_1 && (\alpha_1, \alpha_2 \text{ commute, also } \alpha_1 \bowtie \alpha_2) \\
 \alpha_1 \xrightarrow{\rightarrow} \alpha_2 &\triangleq \alpha_1 \circ \alpha_2 \subseteq \alpha_2 \circ \alpha_1 && (\alpha_1 \text{ right-commutes with } \alpha_2) \\
 \alpha_1 \xrightarrow{\leftarrow} \alpha_2 &\triangleq \alpha_1 \circ \alpha_2 \supseteq \alpha_2 \circ \alpha_1 && (\alpha_1 \text{ left-commutes with } \alpha_2)
 \end{aligned}$$

$$\forall \sigma_1, \sigma_2, \sigma_3 : \begin{array}{c} \sigma_1 \\ \downarrow \stackrel{\alpha_1}{\circlearrowleft} \\ \sigma_2 \xrightarrow{\alpha_2} \sigma_3 \end{array} \Rightarrow \exists \sigma_4 : \begin{array}{c} \sigma_1 \xrightarrow{\alpha_2} \sigma_4 \\ \downarrow \stackrel{\alpha_1}{\circlearrowleft} \\ \sigma_2 \xrightarrow{\alpha_2} \sigma_3 \end{array} \quad (1) \quad \forall \sigma_1, \sigma_2, \sigma_3 : \begin{array}{c} \sigma_1 \xrightarrow{\alpha_2} \sigma_3 \\ \downarrow \stackrel{\alpha_1}{\circlearrowleft} \\ \sigma_2 \end{array} \Rightarrow \exists \sigma_4 : \begin{array}{c} \sigma_1 \xrightarrow{\alpha_2} \sigma_4 \\ \downarrow \stackrel{\alpha_1}{\circlearrowleft} \\ \sigma_2 \xrightarrow{\alpha_2} \sigma_4 \end{array} \quad (2)$$

Left/right commutativity allows actions to be prioritized/delayed over other actions without affecting the end state. Equation 1 illustrates this by quantifying of the states: Action  $\alpha_1$  right-commutes with  $\alpha_2$ , and vice versa  $\alpha_2$  left-commutes with  $\alpha_1$ . Full commutativity ( $\bowtie$ ) always allows both delay and prioritization for any serial execution of  $\alpha_1, \alpha_2$ , while strong commutativity only demands full commutativity when both actions are simultaneously enabled, as shown in Eq. 2 for deterministic actions  $\alpha_1/\alpha_2$  (Eq. 2 is only for an intuition and does not illustrate the non-deterministic case, which is covered by  $\leftrightarrow$ ). Left/right/strong *dependence* implies lack of left/right/strong commutativity, e.g.:  $\alpha_1 \not\bowtie \alpha_2$ .

Note that typically:  $\forall i, \alpha, \beta \in A_i: \alpha \not\bowtie \beta$  due to e.g. a shared program counter. Also note that if  $\alpha_1 \overrightarrow{\bowtie} \alpha_2$ , then  $\alpha_1$  never enables  $\alpha_2$ , while strong commutativity implies that neither  $\alpha$  disables  $\beta$ , nor vice versa.

A lock (/unlock) operation right (/left)-commutes with other locks and unlocks. Indeed, a lock never enables another lock or unlock. Neither do unlocks ever disable other unlocks or locks. In the absence of an unlock however, a lock also attains left-commutativity as it is mutually disabled by other locks. Because of the same disabling property, two locks however do not strongly commute.

Finally, a *necessary enabling set* (NES) of an action  $\alpha$  and a state  $\sigma_1$  is a set of actions that must be executed for  $\alpha$  to become enabled, formally:

$$\forall E \in nes_{\sigma_1}(\alpha), \sigma_1 \xrightarrow{\alpha_1, \dots, \alpha_n} \sigma_2 : \alpha \in \overline{en}(\sigma_1) \wedge \alpha \in en(\sigma_2) \Rightarrow E \cap \{\alpha_1, \dots, \alpha_n\} \neq \emptyset.$$

An example of an action  $\alpha$  with two NESs  $E_1, E_2 \in nes_{\sigma}(\alpha)$  is a command guarded by  $g$  in an imperative language: When  $\alpha \in \overline{en}(\sigma)$ , then either its guard  $g$  does not hold in  $\sigma$ , and  $E_1$  consists of all actions enabling  $g$ , or its program counter is not activated in  $\sigma$ , and  $E_2$  consists of all actions that label the edges immediately before  $\alpha$  in the CFG of the process that  $\alpha$  is part of.

**POR.** POR uses the above relations to find a subset of enabled actions  $por(\sigma) \subseteq en(\sigma)$  sufficient for preserving the property of interest. Commutativity is used to ensure that the sets  $por(\sigma)$  and  $en(\sigma) \setminus por(\sigma)$  commute, while the NES is used to ensure that this mutual commutativity holds *in all future behavior*. The next section explains how stubborn set POR achieves this.

POR gives rise to a CTS  $\widetilde{TS} \triangleq \langle S, \widetilde{T}, A, \sigma_0 \rangle$ ,  $\widetilde{T} \triangleq \{(\sigma, \alpha, \sigma') \in T \mid \alpha \in por(\sigma)\}$ , abbreviated  $\sigma \xrightarrow{\alpha} \sigma'$ . It is indeed reduced, since we have  $\mathcal{R}(\widetilde{TS}) \subseteq \mathcal{R}(TS)$ .

**Transaction reduction.** (Static) transaction reduction was devised by Lipton [38]. It merges multiple sequential statements into one atomic operation, thereby radically reducing the reachable states. An action  $\alpha$  is called a right/left mover if and only if it commutes with actions from all other threads  $j \neq i$ :

$$\xrightarrow{\alpha}_i \overleftrightarrow{\boxtimes} \bigcup_{j \neq i} \longrightarrow_j \quad (\alpha \text{ is a right mover}) \quad \xrightarrow{\alpha}_i \overleftarrow{\boxtimes} \bigcup_{j \neq i} \longrightarrow_j \quad (\alpha \text{ is a left mover})$$

*Both-movers* are transitions that are both left and right movers, whereas *non-movers* are neither. The sequential composition of two movers is also a corresponding mover, and vice versa. Moreover, one may always safely classify an action as a non-mover, although having more movers yields better reductions.

Examples of right-movers are locks, P-semaphores and synchronizing queue operations. Their counterparts; unlock, V-semaphore and enqueue ops, are left-movers. Their behavior is discussed above using locks and unlocks as an example.

Lipton reduction only preserves halting. We present Lamport's [37] version, which preserves safety properties such as  $\Box\varphi$ , i.e.  $\varphi$  is an invariant. Any sequence  $\alpha_1, \dots, \alpha_n$  can be *reduced* to a single action  $\alpha$  s.t.  $\xrightarrow{\alpha}_i = \xrightarrow{\alpha_1}_i \circ \dots \circ \xrightarrow{\alpha_n}_i$  (i.e. a compound statement with the same local behavior), if for some  $1 \leq k < n$ :

**L1** actions before the commit  $\alpha_k$  are right movers:  $\xrightarrow{\alpha_1}_i \circ \dots \circ \xrightarrow{\alpha_{k-1}}_i \overleftrightarrow{\boxtimes} \longrightarrow_{\neq i}$ ,

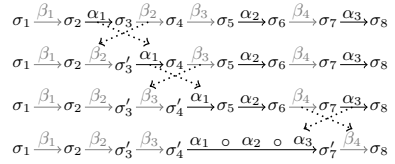
**L2** actions after the commit  $\alpha_k$  are left movers:  $\xrightarrow{\alpha_{k+1}}_i \circ \dots \circ \xrightarrow{\alpha_n}_i \overleftarrow{\boxtimes} \longrightarrow_{\neq i}$ ,

**L3** actions after  $\alpha_1$  do not block:  $\forall \sigma \exists \sigma' : \sigma \xrightarrow{\alpha_1}_i \circ \dots \circ \xrightarrow{\alpha_n}_i \sigma'$ , and

**L4**  $\varphi$  is not disabled by  $\xrightarrow{\alpha_1}_i \circ \dots \circ \xrightarrow{\alpha_{k-1}}_i$ , nor enabled by  $\xrightarrow{\alpha_{k+1}}_i \circ \dots \circ \xrightarrow{\alpha_n}_i$ .



The example (right) shows the evolution of a trace when a reduction with  $n = 3$ ,  $k = 2$  is applied. Actions  $\beta_1, \dots, \beta_4$  are remote. The pre-action  $\alpha_1$  is first moved towards the commit action  $\alpha_2$ . Then the same is done with the post-action  $\alpha_3$ . **L1** resp. **L2** guarantee that the trace's end state  $\sigma_8$  remains invariant, **L3** guarantees its existence and **L4** guarantees that e.g.  $\sigma_4 \notin \varphi \Rightarrow \sigma'_3 \notin \varphi$  and  $\sigma_6 \notin \varphi \Rightarrow \sigma'_7 \notin \varphi$  (preserving invariant violations  $\neg\varphi$  in the reduced system without  $\sigma_4$  and  $\sigma_6$ ). The subsequent section provides a dynamic variant of TR.



### 3 Stubborn Transaction Reduction

The current section gradually introduces stubborn transaction reduction. First, we introduce a stubborn set definition that is parametrized with different commutativity relations. In order to have enough luggage to compare POR to TR in Sect. 4, we elaborate here on various aspects of stubborn POR and compare our definitions to the original stubborn set definitions. We then provide a definition for *dynamic left and right movers*, based on the stubborn set parametrized with left and right commutativity. Finally, we provide a definition of a *transaction system*, show how it is reduced and provide an algorithm to do so. This demonstrates that TR can be made dynamic in the same sense as stubborn sets are dynamic. We focus in the current paper on the preservation of invariants. But since deadlock preservation is an integral part of POR, it is addressed as well.

#### 3.1 Parametrized Stubborn Sets

We use stubborn sets as they have advantages compared to other traditional POR techniques [52, Sect. 4]. We first focus on a basic definition of the stubborn set that only preserves deadlocks. The following version is parametrized (with  $\star$ ).

**Definition 1 ( $\star$ -stubborn sets).** Let  $\star \in \{\leftarrow, \rightarrow, \leftrightarrow\}$ . A set  $B \subseteq A$  is  $\star$ -stubborn in the state  $\sigma$ , written  $st_\sigma^\star(B)$ , if:

- D0**  $en(\sigma) \neq \emptyset \Rightarrow B \cap en(\sigma) \neq \emptyset$  (include an enabled action, if one exists)
- D1**  $\forall \alpha \in B \cap \overline{en}(\sigma): \exists E \in nes_\sigma(\alpha): E \subseteq B$  (for disabled  $\alpha$  include a NES)
- D2**  $\forall \alpha \in B \cap en(\sigma), \beta \not\star \alpha: \beta \in B$  (for enabled  $\alpha$  include  $\star$ -dependent actions)

Notice that a stubborn set  $B$  includes actions disabled in  $\sigma$  to reason over future behavior with **D1**: Actions  $\alpha \in B$  commute with  $\beta \in en(\sigma) \setminus B$  by **D2**, but also with  $\beta' \in en(\sigma')$  for  $\sigma \xrightarrow{\beta} \sigma'$ , since **D1** ensures that  $\beta$  cannot enable any  $\gamma \in B$  (ergo  $\beta' \notin B$ ). Theorem 1 formalizes this. From  $B$ , the reduced system is obtained by taking  $por(\sigma) \triangleq en(\sigma) \cap B$ : It preserves deadlocks. But not all  $\star$ -parametrizations lead to correct reductions w.r.t. deadlock preservation. We therefore briefly relate our definition to the original stubborn set definitions. The above definition yields three interpretations of a set  $B \subseteq A$  for a state  $\sigma$ .

- If  $st_{\sigma}^{\leftrightarrow}(B)$ , then  $B$  coincides with the original *strong* stubborn set [45,46].
- If  $st_{\sigma}^{\leftarrow}(B)$ , then  $B$  approaches the weak stubborn set in [35], a simplified version of [47], except that it lacks a necessary *key action* (from [47, Definition 1.17]).<sup>3</sup>
- If  $st_{\sigma}^{\rightarrow}(B)$ , then  $B$  also may yield an invalid POR, as it would consider two locking operations independent and thus potentially miss a deadlock.

This indicates that POR, unlike TR, cannot benefit from right-commutativity. The consequences of this difference are further discussed in Sect. 4. The strong version of our bare-bone stubborn set definition, on the other hand, is equivalent to the one presented [47] and thus preserves the ‘stubbornness’ property (Theorem 1). If we define *semi-stubbornness*, written  $sst_{\sigma}^{\star}$ , like stubbornness minus the **DO** requirement, then we can prove a similar theorem for semi-stubborn sets (Theorem 2).<sup>4</sup> This ‘stubbornness’ of semi- $\leftarrow$  and semi- $\rightarrow$  stubborn sets is used below to define dynamic movers. First, we briefly return our attention to stubborn POR, recalling how it preserves properties beyond deadlocks and the computation of  $st_{\sigma}$ .

**Theorem 1** ([47]). *If  $B \subseteq A$ ,  $st_{\sigma}^{\leftrightarrow}(B)$  and  $\sigma \xrightarrow{\beta} \sigma'$  for  $\beta \notin B$ , then  $st_{\sigma'}^{\leftrightarrow}(B)$ .*

**Theorem 2.** *If  $B \subseteq A$ ,  $sst_{\sigma}^{\star}(B)$  and  $\sigma \xrightarrow{\beta} \sigma'$  for  $\beta \notin B$ , then  $sst_{\sigma'}^{\star}(B)$  for  $\star \in \{\leftarrow, \leftrightarrow\}$ , as well as for  $\star \in \{\rightarrow\}$  provided that  $\beta$  does not disable a stubborn action, i.e.,  $en(\sigma) \cap B \subseteq en(\sigma') \cap B$ .*

**Stubborn sets for safety properties.** To preserve a safety property such as  $\Box\varphi$  (i.e.  $\varphi$  is invariant), a stubborn set  $B$  ( $st_{\sigma}^{\leftrightarrow}(B) = \text{true}$ ) needs to satisfy two additional requirements [48] called **S** for *safety* and **V** for *visibility*. To express **V**, we denote actions enabling  $\varphi$  with  $A_{\oplus}^{\varphi}$  and those disabling the proposition with  $A_{\ominus}^{\varphi}$ . Those combined form the visible actions:  $A_{vis}^{\varphi} \triangleq A_{\ominus}^{\varphi} \cup A_{\oplus}^{\varphi}$ . For **S**, recall that  $\sigma \xrightarrow{\alpha} \sigma'$  is a reduced transition. *Ignoring states* disrespect **S**.

- S**  $\forall \beta \in en(\sigma): \exists \sigma': \sigma \xrightarrow{\beta} \sigma' \wedge \beta \in por(\sigma')$  (never keep ignoring pruned actions)
- V**  $B \cap en(\sigma) \cap A_{vis}^{\varphi} \neq \emptyset \Rightarrow A_{vis}^{\varphi} \subseteq B$  (either all or no visible, enabled actions)

**Computing stubborn sets and heuristics.** POR is not deterministic as we may compute many different valid stubborn sets for the same state and we can

<sup>3</sup> **DO** is generally not preserved with left-commutativity ( $\star = \leftarrow$ ), as  $\beta \notin B$  may disable  $\alpha \in B$ . Consequently,  $\beta$  may lead to a deadlock. Because POR prunes all  $\beta \notin B$ ,  $st_{\sigma}^{\leftarrow}(B)$  is not a valid reduction (it may prune deadlocks). The key action repairs this by demanding at least one *key action*  $\alpha$ , which strongly commutes, i.e.,  $\forall \beta \in B: \alpha \overset{\sim}{\bowtie} \beta$ , which by virtue of strong commutativity cannot be disabled by any  $\beta \notin B$ .

<sup>4</sup> We will show that semi-stubbornness, i.e.,  $sst_{\sigma}^{\leftarrow}(B)$  (without key), is sufficient for stubborn TR, which may therefore prune deadlocks. Contrarily, invariant-preserving stubborn POR is strictly stronger than the basic stubborn set (see below), and hence also preserves all deadlocks. (This is relevant for the POR/TR comparison in Sect. 4.)

even select different ignoring states to enforce the **S** proviso (i.e. the state  $\sigma'$  in the **S** condition). A general approach to obtain good reductions is to compute a stubborn set with the fewest enabled actions, so that the  $por(\sigma)$  set is the smallest and the most actions are pruned in  $\sigma$ . However, this does not necessarily lead to the best reductions as observed several times [48, 53, 56]. Nonetheless, this is the best heuristic currently available, and it generally yields good results [35].

The  $\forall\exists$ -recursive structure of Definition 1 indicates that establishing the smallest stubborn set is an NP-complete problem, which indeed it is [49]. Various algorithms exist to heuristically compute small stubborn sets [35, 55]. Only the deletion algorithm [55] provides guarantees on the returned sets (that no strict subset of the return set is also stubborn). On the other hand, the guard-based approach [35] has been shown to deliver good reductions in reasonable time.

To implement the **S** proviso, Valmari [47] provides an algorithm [47, Alg. 1.18] that yields the fewest possible ignoring states, runs in linear time and can even be performed on-the-fly, i.e. while generating the reduced transition system. It is based on Tarjan's strongly connected component (SCC) algorithm [44].

The above methods are relevant for stubborn TR as STR also needs to compute ( $\star$ -)stubborn sets and avoid ignoring states (recall L3 from Sect. 2).

### 3.2 Reduced Transaction Systems

TR merges sequential actions into (atomic) transactions and in the process removes interleavings (at the states internal to the transaction) just like POR. We present a dynamic TR that decides to prolong transactions on a per-state basis. We use stubborn sets to identify left and right moving actions in each state. Unlike stubborn set POR, and much like ample-set POR [33], we rely on the process-based action decomposition to identify sequential parts of the system.

Recall that actions in the pre-phase should commute to the right and actions in the post-phase should commute to the left with other threads. We use the notion of stubborn sets to define *dynamic left and right movers* in Eqs. 3 and 4 for  $\langle\sigma, \alpha, \sigma'\rangle \in T_i$ . Both mover definitions are based on semi-stubborn sets. Dynamic left movers are state-based requiring all outgoing local transitions to “move”, whereas right movers are action-based allowing different reductions for various non-deterministic paths. The other technicalities of the definitions stem from the different premises of left and right movability (see Sect. 2). Finally, both dynamic movers exhibit a crucial monotonicity property, similar to previously introduced ‘stubbornness’, as expressed by Lemmas 1 and 2.

$$M_i^{\leftarrow}(\sigma) \triangleq \exists B: sst_{\sigma}^{\leftarrow}(B), B \cap en(\sigma) = A_i \cap en(\sigma) \quad (3)$$

$$M_i^{\rightarrow}(\sigma, \alpha, \sigma') \triangleq \exists B: sst_{\sigma}^{\rightarrow}(B), \alpha \in B, B \cap en(\sigma') \subseteq A_i, B \cap en(\sigma) = \{\alpha\} \quad (4)$$

**Lemma 1.** *The dynamic left-moving property is never remotely disabled, i.e.: if  $M_i^{\leftarrow}(\sigma_1) \wedge i \neq j \wedge \sigma_1 \xrightarrow{\beta}_j \sigma_2$ , then  $M_i^{\leftarrow}(\sigma_2)$ .*

**Lemma 2.** *Dynamic right-movers retain dynamic moveability after moving, i.e.: if  $M_i^{\rightarrow}(\sigma_1, \alpha, \sigma_2) \wedge \sigma_1 \xrightarrow{\alpha}_i \sigma_2 \xrightarrow{\beta}_j \sigma_3$  for  $i \neq j$ , then  $\exists \sigma_1 \xrightarrow{\beta}_j \sigma_4 : M_i^{\rightarrow}(\sigma_4, \alpha, \sigma_3)$ .*

To establish stubborn TR, Definition 2 first annotates the transition system TS with thread-local phase information, i.e. one phase variable for each thread that is only modified by that thread. Phases are denoted with Ext (for transaction external states), Pre (for states in the pre-phase) and Post (for states in the post phase). Because phases now depend on the commutativity established via dynamic movers Eqs. 3 and 4, the reduction (not included in the definition, but discussed below it) becomes dynamic. Lemma 3 follows easily as the definition does not yet enforce the reduction, but mostly ‘decorates’ the transition system.

**Definition 2 (Transaction system).** *Let  $H \triangleq \{\text{Ext}, \text{Pre}, \text{Post}\}^P$  be an array of local phases. The transaction system is CTS  $\text{TS}' \triangleq \langle S', T', A, \sigma'_0 \rangle$  such that:*

$$\begin{aligned}
 S' &\triangleq S \times H, & \sigma'_0 &\triangleq \langle \sigma_0, \text{Ext}^P \rangle \\
 T'_i &\triangleq \{ \langle \langle \sigma, h \rangle, \alpha, \langle \sigma', h' \rangle \rangle \in S' \times A \times S' \mid (\sigma, \alpha, \sigma') \in T_i, \forall j \neq i: h'_j = h_j, \\
 & \left\{ \begin{array}{ll} \text{Pre} & \text{iff } h_i \neq \text{Post} \wedge M_i^{\rightarrow}(\sigma, \alpha, \sigma') \wedge \alpha \notin A_{\ominus}^{\varphi} & (5) \\ \text{Post} & \text{iff } M_i^{\leftarrow}(\sigma') \wedge \text{en}(\sigma') \cap A_i \cap A_{\oplus}^{\varphi} = \emptyset & (6) \\ \text{Ext} & \text{otherwise (or as alternative when Eq. 6 holds)} & (7) \end{array} \right. \\
 & \}
 \end{aligned}$$

**Lemma 3.** *Definition 2 preserves invariants:  $\mathcal{R}(\text{TS}) \models \Box \varphi \Leftrightarrow \mathcal{R}(\text{TS}') \models \Box \varphi$ .*

The conditions in Eqs. 6 and 7 overlap on purpose, allowing us to enforce termination below. The transaction system effectively partitions the state spaces on the phase for each thread  $i$ , i.e.  $\overline{\text{Ext}}_i = \text{Post}_i \cup \text{Pre}_i$  with  $\text{Ext}_i \triangleq \{ \langle \sigma, h \rangle \mid h_i = \text{Ext} \}$ , etc. The definition of  $T'_i$  further ensures three properties:

- A.  $\text{Post}_i$  states do not transit to  $\text{Pre}_i$  states as  $h_i = \text{Post} \Rightarrow h'_i \neq \text{Pre}$  by Eq. 5.
- B. Transitions ending in  $\text{Pre}_i$  are dynamic right movers not disabling  $\varphi$  by Eq. 5.
- C. Transitions starting in  $\text{Post}_i$  are dynamic left movers not enabling  $\varphi$  by Eq. 6.

Thereby  $T'_i$  implements the (syntactic) constraints from Lipton’s TR (see Sect. 2) dynamically in the transition system, except for **L3**. Let  $\xrightarrow{'}_i \triangleq \{ \langle q, q' \rangle \mid \langle q, \alpha, q' \rangle \in T'_i \}$ . Next, Theorem 3 defines the reduced transaction system (RTS), based primarily on the  $\hookrightarrow$  transition relation that only allows a thread  $i$  to transit when all other threads are in an external state, thus eliminating interleavings ( $\rightsquigarrow$  additionally skips internal states). The theorem concludes that invariants are preserved given that a termination criterium weaker than **L3** is met: All  $\text{Post}_i$  must reach an  $\text{Ext}_i$  state. Monotonicity of dynamic movers plays a key role in its proof.

---

**Algorithm 1.** Algorithm reducing a CTS to an RTS using  $T'_i$  from Definition 2.
 

---

<pre> 1: <math>V_1, V_2, Q_1, Q_2 : S'</math> 2: <b>proc</b> SEARCH(TS <math>\triangleq \langle S, T, A, \sigma_0 \rangle</math>) 3:   <math>Q_1 := \{ \langle \sigma_0, \text{Ext}^P \rangle \}</math> 4:   <math>V_1 := \emptyset</math> 5:   <b>while</b> <math>Q_1 \neq \emptyset</math> <b>do</b> 6:     <math>Q_1 := Q_1 \setminus \{ \langle \sigma, h \rangle \}</math> <b>for</b> <math>\langle \sigma, h \rangle \in Q_1</math> 7:     <math>V_1 := V_1 \cup \{ \langle \sigma, h \rangle \}</math> 8:     <b>assert</b> <math>(\forall i : h_i = \text{Ext})</math> 9:     <b>for</b> <math>i \in P</math> <b>do</b> 10:      TRANSACTION(T, <math>\langle \sigma, h \rangle</math>, <math>i</math>) 11:     <b>assert</b> <math>(V_1 = \mathcal{R}(\tilde{\text{TS}}))</math> 12: <b>function</b> SCCROOT(<math>q, i</math>) 13:   <b>return</b> <math>q</math> is a root of bottom SCC <math>C</math>       <b>s.t.</b> <math>C \subseteq \text{Post}_i \wedge C \subseteq V_2</math> </pre>	<pre> 14: <b>proc</b> TRANSACTION(T, <math>\langle \sigma, h \rangle</math>, <math>i</math>) 15:   <math>Q_2 := \{ \langle \sigma, h \rangle \}</math> 16:   <math>V_2 := \emptyset</math> 17:   <b>while</b> <math>Q_2 \neq \emptyset</math> <b>do</b> 18:     <math>Q_2 := Q_2 \setminus \{ \langle \sigma, h \rangle \}</math> <b>for</b> <math>\langle \sigma, h \rangle \in Q_2</math> 19:     <math>V_2 := V_2 \cup \{ \langle \sigma, h \rangle \}</math> 20:     <b>for</b> <math>\langle \sigma, \alpha, \sigma' \rangle \in T_i</math> <b>do</b> 21:       <b>let</b> <math>h'</math> <b>s.t.</b> <math>\langle \langle \sigma, h \rangle, \alpha, \langle \sigma', h' \rangle \rangle \in T'_i</math> 22:       <b>if</b> SCCROOT(<math>\langle \sigma', h' \rangle</math>, <math>i</math>) <b>then</b> 23:         <math>h'_i := \text{Ext}</math> 24:         <b>if</b> <math>\langle \sigma', h' \rangle \not\sqsubseteq V_1 \cup V_2 \cup Q_1 \cup Q_2</math> <b>then</b> 25:           <math>Q_2 := Q_2 \cup \{ \langle \sigma', h' \rangle \}</math> 26:         <b>if</b> <math>h'_i = \text{Ext} \wedge \langle \sigma', h' \rangle \notin V_1 \cup Q_1</math> <b>then</b> 27:           <math>Q_1 := Q_1 \cup \{ \langle \sigma', h' \rangle \}</math> </pre>
---	---

---

**Theorem 3 (Reduced Transaction System (RTS)).** We define for all  $i$ :

$$\begin{aligned} \hookrightarrow_i &\triangleq (\cup_{j \neq i} \text{Ext}_j) // \longrightarrow'_i && (i \text{ only transits when all } j \text{ are external}) \\ \rightsquigarrow_i &\triangleq \text{Ext}_i // (\hookrightarrow_i \setminus \overline{\text{Ext}_i})^* \hookrightarrow_i // \text{Ext}_i && (\text{skip internal states transition relation}) \end{aligned}$$

The RTS is a CST  $\tilde{\text{TS}} \triangleq \langle S', \{ \langle q, \alpha_i, q' \rangle \mid q \rightsquigarrow_i^{\alpha_i} q' \}, A, \sigma'_0 \rangle$ . Now, provided that  $\forall \sigma \in \text{Post}_i : \exists \sigma' \in \text{Ext}_i : \sigma \hookrightarrow_i^* \sigma'$ , we have  $\mathcal{R}(\text{TS}) \models \Box \varphi \iff \mathcal{R}(\tilde{\text{TS}}) \models \Box \varphi$ .

The following algorithm generates the RTS  $\tilde{\text{TS}}$  of Theorem 3 from a TS. The state space search is split into two: One main search, which only processes external states ( $\bigcap_i \text{Ext}_i$ ), and an additional search (TRANSACTION) which explores the transaction for a single thread  $i$ . Only when the transaction search encounters an external state, it is propagated back to the queue  $Q_1$  of the main search, provided it is new there (not yet in  $V_1$ , which is checked at Line 26). The transaction search terminates early when an internal state  $q$  is found to be subsumed by an external state already encountered in the outer search (see the  $q \not\sqsubseteq V_1$  check at Line 24). Subsumption is induced by the following order on phases, which is lifted to states and sets of states  $X \subseteq S'$ :  $\text{Pre} \sqsubset \text{Post} \sqsubset \text{Ext}$  with  $a \sqsubseteq b \iff a = b \vee a \sqsubset b$ ,  $\langle \sigma, h \rangle \sqsubseteq \langle \sigma', h' \rangle \iff \sigma = \sigma' \wedge \forall i : h_i \sqsubseteq h'_i$ , and  $q \sqsubseteq X \iff \forall q' \in X : q \sqsubseteq q'$  (for  $q = \langle \sigma, h \rangle$ ).

Termination detection is implemented using Tarjan's SCC algorithm as in [47]. We chose not to obfuscate the search with the rather intricate details of that algorithm. Instead, we assume that there is a function SCCROOT which identifies a unique *root* state in each bottom SCC composed solely of post-states. This state is then made external on Line 23 fulfilling the premise of Theorem 3 ( $\forall \sigma \in \text{Post}_i : \exists \sigma' \in \text{Ext}_i : \sigma \hookrightarrow_i^* \sigma'$ ). Combined with Lemma 3 this yields Theorem 4.

**Theorem 4.** Algorithm 1 computes  $\mathcal{R}(\tilde{\text{TS}})$  s.t.  $\mathcal{R}(\text{TS}) \models \Box \varphi \iff \mathcal{R}(\tilde{\text{TS}}) \models \Box \varphi$ .

Finally, while the transaction system exponentially blows up the number of syntactic states ( $\neq$  reachable states) by adding local phase variables, the reduction completely hides this complexity as Theorem 5 shows. Therefore, as soon as the reduction succeeds in removing a single state, we have by definition

that  $|\mathcal{R}(\text{TS})| < |\mathcal{R}(\widetilde{\text{TS}})|$ . Theorem 5 also allows us to simplify the algorithm by storing transition system states  $S$  instead of transaction system states  $S'$  in  $V_1$  and  $Q_1$ .

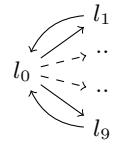
**Theorem 5.** *Let  $N \triangleq \cap_i \text{Ext}_i$ . We have  $|N| = |S|$  and  $\mathcal{R}(\widetilde{\text{TS}}) \subseteq \mathcal{R}(\text{TS})$ .*

### 4 Comparison Between TR and POR

Stubborn TR (STR) is dynamic in the same sense as stubborn POR, allowing for a better comparison of the two. To this end, we discuss various example types of systems that either TR or POR excel at. As a basis, consider a completely independent system with  $p$  threads of  $n - 1$  operations each. Its state space has  $n^p$  states. TR can reduce a state space to  $2^p$  states whereas POR yields  $n * p$  states. The question is however also which kinds of systems are realistic and whether the reductions can be computed precisely and efficiently.

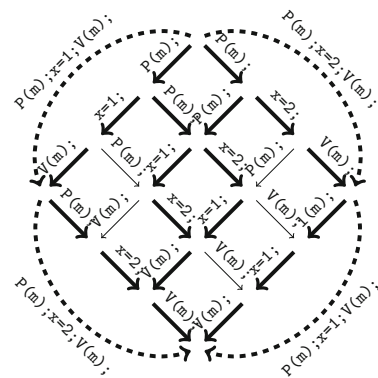
**High parallelism vs Long sequences of local transitions.** POR has an advantage when  $p \gg n$  being able to yield exponential reductions. Though e.g. thread-modular verification [11, 39] may become more attractive in those cases. Software verification often has to deal with many sequential actions benefitting STR, especially when VM languages such as LLVM are used [24].

**Non-determinism.** In the pre-phase, TR is able to individually reduce mutually non-deterministic transitions of one thread due to Eq. 5, which contrary to Eq. 6 considers individual actions of a thread. Consider the example on the right. It represents a system with nine non-deterministic steps in a loop. Assume one of them never commutes, but the others commute to the right. Stubborn TR is able to reduce all paths through the loop over only the right-movers, even if they constantly yield new states (and interleavings).



**Left and right movers.** While stubborn POR can handle left-commutativity using additional restrictions, STR can benefit from right-commutativity in the pre-phase and from left-commutativity in the post-phase. E.g., P/V-semaphores are right/left-movers (see Sect. 2). Figure 2 shows a system with ideal reduction using TR, and none with stubborn set POR.

Table 1 provides various synchronization constructs and their movability. Thread create & join have not been classified before.



**Fig. 2.** State space of  $P(m); x=1; V(m); \parallel P(m); x=2; V(m)$  and POR (thick lines) and TR (dashed lines).

**Deadlocks.** POR preserves all deadlocks, even when irrelevant to the property. TR does not preserve deadlocks at all, potentially allowing for better reductions preserving invariants. The following example

deadlocks because of an invalid

**Table 1.** Movability of commonly used synchronization mechanisms

<code>pthread.create</code>	As this can be modeled with a mutex that is guarding the thread's code and is initially set to locked, the <code>create</code> -call is an unlock and thus a left-mover
<code>pthread.join</code>	Using locking similar to <code>create</code> , <code>join</code> becomes a lock and thus a right-mover
Re-entrant locks	Right/left movers [13]
Wait/notify/notifyAll	Can all three be split into right and left moving parts [13]

locking order. TR can still reduce the example to four states, creating maximal transactions. On the other hand, POR must explore the deadlock.

$1(m1); 1(m2); x=1; u(m1); u(m2); \parallel 1(m2); 1(m1); x=2; u(m1); u(m2);$

**Processes.** STR retains the process-based definition from its ancestors [38], while stubborn POR can go beyond process boundaries to improve reductions and even supports process algebras [35, 51]. In early attempts to solve the open problem of a process-less STR definition, we observed that inclusion of all actions in a transaction could cause the entire state space search to move to the SEARCH-TRANSACTION function.

**Tractability and heuristics.** The STR algorithm can fix the set of stubborn transitions to those in the same thread (see definitions of  $M_\alpha^*$ ). This can be exploited in the deletion algorithm by fixing the relevant transitions (see the incomplete minimization approach [55]). If the algorithm returns a set with other transitions, then we know that no transaction reduction is possible as the returned set is subset-minimal [35, Th. 1]. The deletion algorithm runs in polynomial time (in the order of  $|A|^4$  [48]), hence also stubborn TR also does (on a per-state basis). Stubborn set POR, however, is NP-complete as it has to consider all subsets of actions. Moreover, a small stubborn set is merely a heuristic for optimal reductions [49] as discussed in Sect. 3.1.

**Known unknowns.** We did not consider other properties such as full safety, LTL and CTL. For CTL, POR can no longer reduce to non-trivial subsets because of the CTL proviso [17] (see [51] for support of non-deterministic transitions, like in stubborn TR). TR for CTL is an open problem.

While TR can split visibility in enabling (in the pre-phase) and disabling (in the post-phase), POR must consider both combined. POR moreover must compute the ignoring proviso over the entire state space while TR only needs to consider post-phases and thread-local steps.

The ignoring proviso [5, 10, 50] in POR tightly couples the possible reductions per state to the role the state plays in the entire reachability graph. This lack of locality adds an extra obstacle to the parallelization of the model checking procedure. Early results make compromises in the obtained reductions [4]. Recent results show that reductions do not have to be affected negatively even with high amounts of parallelism [36], however these results have not yet been achieved for distributed systems. TR reduction on the other hand, offers plenty of parallelization opportunities, as each state in the out search can be handed off to a separate process.

## 5 Experiments

We implemented stubborn transaction reduction (STR) of Algorithm 1 in the open source model checker LTSMIN<sup>5</sup> [31], using a modified deletion algorithm to establish optimal stubborn sets in polynomial time (as discussed in Sect. 4). The implementation can be found on GitHub.<sup>6</sup> LTSMIN has a front-end for PROMELA models, which is on par with the SPIN model checker [26] performance-wise [54]. Unlike SPIN, LTSMIN does not implement dynamic commutativity specifically for queues [27], but because it splits queue actions into a separate action for each cell [54], a similar result is achieved by virtue of the stubborn set condition **D1** in Sect. 3.1. This benefits both its POR and STR implementation.

We compare STR against (static) TR from Sect. 2. We also compare STR against the stubborn set POR in LTSMIN, which was shown to consistently outperform SPIN’s ample set [27] implementation in terms of reductions, but with worse run-times due to the more elaborate stubborn set algorithms (a factor 2–4) [35]. (We cannot compare with [25] due to the different input formats of VVT [24] and LTSMIN.) Table 2 shows the models that we considered and their normal (unreduced) verification times in LTSMIN. We took all models from [35] that contained an assertion. The inputs include mutual exclusion algorithms (`peterson`), protocol implementations (`i-protocol`, `BRP`, `GARP`, `X509`), a lockless queue (`MSQ`) and controllers (`SMCS`, `SMALL1`, `SMALL2`).

LTSMIN runs with STR were configured according to the command line:

```
prom2lts-mc --por=str --timeout=3600 -n --action=assert m.spins
```

The option `--por=tr` enables the static TR instead. We also run all models in SPIN in order to compare against the ample set’s performance. SPIN runs were configured according to the following command lines:

```
cc -O3 -DNOFAIR -DREDUCE -DNOBOUNDCHECK -DNOCOLLAPSE -DSAFETY -DMEMLIM=100000 -o pan pan.c
./pan -m10000000 -c0 -n -w20
```

**Table 2.** Models and their verification times in LTSMIN. Time in sec. and memory use in MB. State/transition counts are the same in both LTSMIN and SPIN.

	SPIN/LTSMIN		LTSMIN	
	states	transitions	time	mem
<code>Peterson5</code>	829909270	3788955584	4201.	6556.
<code>GARP</code>	48363145	247135869	88.34	369.8
<code>i-Prot.2</code>	13168183	44202271	22.99	102.8
<code>i-Prot.0</code>	9798465	45932747	19.58	75.2
<code>Peterson4</code>	3624214	13150952	7.36	28.5
<code>BRP</code>	2812740	6166206	4.59	26.4
<code>MSQ</code>	994819	3198531	4.41	12.1
<code>i-Prot.3</code>	327358	978579	0.79	2.8
<code>i-Prot.4</code>	78977	169177	0.19	0.8
<code>Small11</code>	36970	163058	0.14	0.3
<code>X.509</code>	9028	35999	0.03	0.1
<code>Small12</code>	7496	32276	0.08	0.1
<code>SMCS</code>	2909	10627	0.01	0.1

<sup>5</sup> <http://fmt.cs.utwente.nl/tools/ltsmin/>.

<sup>6</sup> <https://github.com/alaarman/ltsmin/commits/tr>.



**Table 3.** Reduction runs of TR, Stubborn TR (STR) and Stubborn POR (SPOR). Reductions of states  $|S|$  and transitions  $|T|$  are given in percentages (reduced state space/original state space), runtimes in sec. and memory use in MB. The lowest reductions (in number of states) and the runtimes are highlighted in bold.

	TR (LTSMIN)				STR (LTSMIN)				SPOR (LTSMIN)				Ampe set (SPIN)			
	$ S $	$ T $	time	mem	$ S $	$ T $	time	mem	$ S $	$ T $	time	m	$ S $	$ T $	time	mem
Peterson5	0.5	0.3	<b>6.11</b>	33.0	<b>0.4</b>	0.3	74.01	29.5	3.1	0.9	316.10	209.8	5.2	1.9	42.30	2463.
GARP	100	100	266.21	369.8	<b>1.4</b>	1.5	776.53	5.2	3.6	1.5	19.83	13.5	7.6	3.7	<b>6.27</b>	289.1
i-Prot.2	<b>2.1</b>	2.4	<b>3.46</b>	2.2	<b>2.1</b>	2.4	4.87	2.2	20.2	11.9	13.32	21.7	26.1	17.6	4.33	246.9
i-Prot.0	100	100	56.71	75.2	<b>12.8</b>	12.5	148.78	9.7	32.1	17.2	214.93	24.3	15.7	10.5	<b>2.56</b>	132.2
Peterson4	<b>1.3</b>	1.0	0.36	0.5	<b>1.3</b>	1.0	0.85	0.5	7.3	2.7	4.24	2.4	14.7	6.8	<b>0.24</b>	28.9
BRP	100	100	9.59	26.4	47.6	36.9	6.38	12.6	100	100	90.31	26.4	<b>9.2</b>	6.0	<b>0.18</b>	22.2
MSQ	66.0	65.0	5.5	8.2	<b>22.9</b>	21.5	14.90	3.0	52.1	29.1	12.14	6.5	80.4	46.6	<b>1.03</b>	200.9
i-Prot.3	8.0	7.4	0.19	0.2	<b>8.0</b>	7.4	0.24	0.2	20.7	10.4	0.94	0.6	27.0	16.5	<b>0.06</b>	5.8
i-Prot.4	25.1	27.2	0.14	0.2	<b>25.0</b>	27.1	0.18	0.2	45.2	31.5	0.54	0.4	50.4	37.1	<b>0.03</b>	2.8
Small1	8.9	18.0	0.03	n/a	<b>6.7</b>	13.6	0.07	n/a	31.2	17.7	0.18	0.1	48.4	45.1	<b>0.01</b>	0.9
X.509	93.8	94.1	0.07	0.1	19.3	16.7	0.06	n/a	<b>7.8</b>	3.7	0.03	n/a	67.5	34.3	<b>0.01</b>	1.1
Small12	11.6	21.0	0.01	n/a	<b>8.7</b>	15.8	0.01	n/a	35.0	19.8	0.04	n/a	48.3	43.8	<b>0.01</b>	0.4
SMCS	100	100	0.05	0.1	26.1	19.6	0.09	n/a	<b>12.5</b>	5.3	0.03	n/a	41.1	19.6	<b>0.01</b>	0.7

Table 3 shows the benchmark results. We observe that STR often surpasses POR (stubborn and ample sets) in terms of reductions. Its runtimes however are inferior to those of the ample set in SPIN. This is likely because we use the precise deletion algorithm, which decides the optimal reduction for STR: STR is the only algorithm of the four that does not use heuristics. The higher runtimes of STR are often compensated by the better reductions it obtains.

Only three models demonstrate that POR can yield better reductions (BRP, smcs and X.509). This is perhaps not surprising as these models do not have massive parallelism (see Sect. 4). It is however interesting to note that GARP contains seven threads. We attribute the good reductions of STR mostly to its ability to skip internal states. SPIN’s ample set only reduces the BRP better than LTSMIN’s stubborn POR and STR. In this case, we found that LTSMIN too eagerly identifies half of the actions of both models as visible.

**Validation.** Validation of TR is harder than of POR. For POR, we usually count deadlocks, as all are preserved, but TR might actually prune deadlocks and error states (while preserving the invariant as per Theorem 4). We therefore tested correctness of our implementation by implementing methods that check the validity of the returned semi-sturbborn sets. Additionally, we maintained counters for the length of the returned transactions and inspected the inputs to confirm validity of the longest transactions.

## 6 Related Work

Lipton’s reduction was refined multiple times [6, 7, 21, 37, 43]. Flanagan et al. [11, 15] and Qadeer et al. [13, 14, 16] have most recently developed transactions and found various applications. The reduction theorem used to prove

the theorems in the current paper comes from our previous work [25], which in turn is a generalized version of [13]. Our generalization allows the direct support of dynamic transactions as already demonstrated for symbolic model checking with IC3 in [25]. Despite a weaker theorem, Qadeer and Flanagan [13] can also dynamically grow transactions by doing iterative refinement over the state space exploration. This contrasts our approach, which instead allows on-the-fly adaptation of movability (within a single exploration). Moreover, [13] bases dynamic behavior on exclusive access to variables, whereas our technique can handle any kind of dependency captured by the general stubborn set POR relations.

Cartesian POR [23] is a form of Lipton reduction that builds transactions during the exploration, but does not exploit left/right commutativity. The leap set method [42] treats disjoint reduced sets in the same state as truly concurrent and executes them as such: The product of the different disjoint sets is executed from the state, which entails that sequences of actions are executed from the state. This is where the similarity with the TR ends, because in TR the sequences are formed by sequential actions, whereas in leap sets they consist of concurrent actions, e.g., actions from different processes. Recently, trace theory has been generalized to include ‘steps’ by Ryszard et al. [28]. We believe that this work could form a basis to study leap sets and TR in more detail.

Various classical POR works were mentioned, e.g. [20, 33, 46]. How ‘persistent sets’ [20]/‘ample sets’ [33] relate to stubborn set POR is explained in [52, Sect. 4]. Sleep sets [19] form an orthogonal approach, but in isolation only reduce the number of transitions. Dwyer et al. [8] propose dynamic techniques for object-oriented programs. Completely dynamic approaches exist [12, 32]. Recently, even optimal solutions were found [1, 2, 41]. These approaches are typically stateless however, although still succeed in pruning converging paths sometimes (e.g., [41]). Others aim at making dependency more dynamic [18, 27, 34].

Symbolic POR can be more static for reasons discussed in Footnote 2, e.g., [3]. Therefore, Grumberg et al. [22] present underapproximation-widening, which iteratively refines an under-approximated encoding of the system. In their implementation, interleavings are constrained to achieve the under-approximation. Because refinement is done based on verification proofs, irrelevant interleavings will never be considered. Other relevant dynamic approaches are peephole and monotonic POR by Wang et al. [30, 57]. Like sleep sets [20], however, these methods only reduce the number of transitions. While a reducing transitions can speed up symbolic approaches by constraining the transition relation, it is not useful for enumerative model checking, which is strongly limited by the amount of unique states that need to be stored in memory.

Kahlon et al. [29] do not implement transactions, but encode POR for symbolic model checking using SAT. The “sensitivity” to locks of their algorithm can be captured in traditional stubborn sets as well by viewing locks as normal “objects” (variables) with guards, resulting in the subsumption of the “might-be-the-first-to-interfere-modulo-lock-acquisition” relation [29] by the “might-be-the-first-to-interfere” relation [29], originally from [20].

Elmas et al. [9] propose dynamic reductions for type systems, where the invariant is used to weaken the mover definition. They also support both right and left movers, but do automated theorem proving instead of model checking.

## 7 Conclusion

We presented a more dynamic version of transaction reduction (TR) based on techniques from stubborn set POR. We analyzed several scenarios for which either of the two approaches has an advantage and also experimentally compared both techniques. We conclude that TR is a valuable alternative to POR at least for systems with a relatively low amount of parallelism.

Both in theory and practice, TR showed advantages to POR, but vice versa as well. Most strikingly, TR is able to exploit various synchronization mechanisms in typical parallel programs because of their left and right commutativity. While not preserving deadlocks, its reductions can benefit from omitting them. These observations are supported by experiments that show better reductions than a comparably dynamic POR approach for systems with up to 7 threads. We observe that the combination POR and TR is an open problem.

## References

1. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: POPL, pp. 373–384. ACM (2014)
2. Albert, E., Arenas, P., de la Banda, M.G., Gómez-Zamalloa, M., Stuckey, P.J.: Context-sensitive dynamic partial order reduction. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 526–543. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-63387-9\\_26](https://doi.org/10.1007/978-3-319-63387-9_26)
3. Alur, R., Brayton, R.K., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-order reduction in symbolic state space exploration. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 340–351. Springer, Heidelberg (1997). [https://doi.org/10.1007/3-540-63166-6\\_34](https://doi.org/10.1007/3-540-63166-6_34)
4. Barnat, J., Brim, L., Ročkait, P.: Parallel partial order reduction with topological sort proviso. In: SEFM, pp. 222–231. IEEE (2010)
5. Bošnački, D., Holzmann, G.J.: Improving spin’s partial-order reduction for breadth-first search. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 91–105. Springer, Heidelberg (2005). [https://doi.org/10.1007/11537328\\_10](https://doi.org/10.1007/11537328_10)
6. Cohen, E., Lamport, L.: Reduction in TLA. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 317–331. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0055631>
7. Doepfner Jr., T.W.: Parallel program correctness through refinement. In: POPL, pp. 155–169. ACM (1977)
8. Dwyer, M., et al.: Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *FMSD* **25**(2–3), 199–240 (2004)
9. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: POPL, pp. 2–15. ACM (2009)

10. Evangelista, S., Pajault, C.: Solving the ignoring problem for partial order reduction. *STTT* **12**, 155–170 (2010)
11. Flanagan, C., Freund, S.N., Qadeer, S.: Thread-modular verification for shared-memory programs. In: Le Métayer, D. (ed.) *ESOP 2002*. LNCS, vol. 2305, pp. 262–277. Springer, Heidelberg (2002). [https://doi.org/10.1007/3-540-45927-8\\_19](https://doi.org/10.1007/3-540-45927-8_19)
12. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: *POPL*, vol. 40, no. 1, pp. 110–121. ACM (2005)
13. Flanagan, C., Qadeer, S.: Transactions for software model checking. *ENTCS* **89**(3), 518–539 (2003)
14. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: *PLDI*, pp. 338–349. ACM (2003)
15. Flanagan, C., Qadeer, S.: Types for atomicity. In: *SIGPLAN Notices*, vol. 38(3), pp. 1–12, January 2003
16. Freund, S.N., Qadeer, S.: Checking concise specifications for multithreaded software. *J. Object Technol.* **3**, 81–101 (2004)
17. Gerth, R., Kuiper, R., Peled, D., Penczek, W.: A partial order approach to branching time logic model checking. In: *TCS*, pp. 130–139. IEEE (1995)
18. Godefroid, P., Pirottin, D.: Refining dependencies improves partial-order verification methods (extended abstract). In: Courcoubetis, C. (ed.) *CAV 1993*. LNCS, vol. 697, pp. 438–449. Springer, Heidelberg (1993). [https://doi.org/10.1007/3-540-56922-7\\_36](https://doi.org/10.1007/3-540-56922-7_36)
19. Godefroid, P., Wolper, P.: Using partial orders for the efficient verification of deadlock freedom and safety properties. *FMSD* **2**, 149–164 (1993)
20. Godefroid, P. (ed.): *Partial-Order Methods for the Verification of Concurrent Systems*. LNCS, vol. 1032. Springer, Heidelberg (1996). <https://doi.org/10.1007/3-540-60761-7>
21. Gribomont, E.P.: Atomicity refinement and trace reduction theorems. In: Alur, R., Henzinger, T.A. (eds.) *CAV 1996*. LNCS, vol. 1102, pp. 311–322. Springer, Heidelberg (1996). [https://doi.org/10.1007/3-540-61474-5\\_79](https://doi.org/10.1007/3-540-61474-5_79)
22. Grumberg, O., et al.: Proof-guided underapproximation-widening for multi-process systems. In: *POPL*, pp. 122–131. ACM (2005)
23. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. In: Bošnački, D., Edelkamp, S. (eds.) *SPIN 2007*. LNCS, vol. 4595, pp. 95–112. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-73370-6\\_8](https://doi.org/10.1007/978-3-540-73370-6_8)
24. Günther, H., Laarman, A., Weissenbacher, G.: Vienna verification tool: IC3 for parallel software. In: Chechik, M., Raskin, J.-F. (eds.) *TACAS 2016*. LNCS, vol. 9636, pp. 954–957. Springer, Heidelberg (2016). [https://doi.org/10.1007/978-3-662-49674-9\\_69](https://doi.org/10.1007/978-3-662-49674-9_69)
25. Günther, H., Laarman, A., Sokolova, A., Weissenbacher, G.: Dynamic reductions for model checking concurrent software. In: Bouajjani, A., Monniaux, D. (eds.) *VMCAI 2017*. LNCS, vol. 10145, pp. 246–265. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-52234-0\\_14](https://doi.org/10.1007/978-3-319-52234-0_14)
26. Holzmann, G.J.: The model checker SPIN. *IEEE TSE* **23**, 279–295 (1997)
27. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: *IFIP WG6.1 ICFDT VII*, pp. 197–211. Chapman & Hall Ltd. (1995)
28. Janicki, R., Kleijn, J., Koutny, M., Mikulski, L.: Step traces. *Acta Inform.* **53**(1), 35–65 (2016)
29. Kahlon, V., Gupta, A., Sinha, N.: Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In: Ball, T., Jones, R.B. (eds.) *CAV 2006*. LNCS, vol. 4144, pp. 286–299. Springer, Heidelberg (2006). [https://doi.org/10.1007/11817963\\_28](https://doi.org/10.1007/11817963_28)

30. Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: an optimal symbolic partial order reduction technique. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 398–413. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-02658-4\\_31](https://doi.org/10.1007/978-3-642-02658-4_31)
31. 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](https://doi.org/10.1007/978-3-662-46681-0_61)
32. Kastenbergh, H., Rensink, A.: Dynamic partial order reduction using probe sets. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 233–247. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-85361-9\\_21](https://doi.org/10.1007/978-3-540-85361-9_21)
33. Katz, S., Peled, D.: An efficient verification method for parallel and distributed programs. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1988. LNCS, vol. 354, pp. 489–507. Springer, Heidelberg (1989). <https://doi.org/10.1007/BFb0013032>
34. Katz, S., Peled, D.: Defining conditional independence using collapses. *Theor. Comput. Sci.* **101**(2), 337–359 (1992)
35. Laarman, A.W., Pater, E., van de Pol, J.C., Hansen, H.: Guard-based partial-order reduction. In: STTT, pp. 1–22 (2014)
36. Laarman, A., Wijs, A.: Partial-order reduction for multi-core LTL model checking. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 267–283. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-13338-6\\_20](https://doi.org/10.1007/978-3-319-13338-6_20)
37. Lamport, L., Schneider, F.B.: Pretending atomicity. Technical report, Cornell University (1989)
38. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. *Commun. ACM* **18**(12), 717–721 (1975)
39. Malkis, A., Podelski, A., Rybalchenko, A.: Thread-modular verification is Cartesian abstract interpretation. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) ICTAC 2006. LNCS, vol. 4281, pp. 183–197. Springer, Heidelberg (2006). [https://doi.org/10.1007/11921240\\_13](https://doi.org/10.1007/11921240_13)
40. Papadimitriou, C.: *The Theory of Database Concurrency control*. Principles of Computer Science Series. Computer Science Press, Rockville (1986)
41. Rodríguez, C., et al.: Unfolding-based partial order reduction. In: CONCUR. LIPIcs, vol. 42, pp. 456–469. Leibniz-Zentrum fuer Informatik (2015)
42. Van Der Schoot, H., Ural, H.: An improvement of partial-order verification. *Softw. Test. Verif. Reliab.* **8**(2), 83–102 (1998)
43. Stoller, S.D., Cohen, E.: Optimistic synchronization-based state-space reduction. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 489–504. Springer, Heidelberg (2003). [https://doi.org/10.1007/3-540-36577-X\\_36](https://doi.org/10.1007/3-540-36577-X_36)
44. Tarjan, R.: Depth-first search and linear graph algorithms. *SIAM J. Comput.* **1**(2), 146–160 (1972)
45. Valmari, A.: Error detection by reduced reachability graph generation. In: APN, pp. 95–112 (1988)
46. Valmari, A.: Eliminating redundant interleavings during concurrent program verification. In: Odijk, E., Rem, M., Syre, J.-C. (eds.) PARLE 1989. LNCS, vol. 366, pp. 89–103. Springer, Heidelberg (1989). [https://doi.org/10.1007/3-540-51285-3\\_35](https://doi.org/10.1007/3-540-51285-3_35)
47. 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](https://doi.org/10.1007/3-540-53863-1_36)

48. Valmari, A.: The state explosion problem. In: Reisig, W., Rozenberg, G. (eds.) ACPN 1996. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998). [https://doi.org/10.1007/3-540-65306-6\\_21](https://doi.org/10.1007/3-540-65306-6_21)
49. Valmari, A., Hansen, H.: Can stubborn sets be optimal? In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 43–62. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-13675-7\\_5](https://doi.org/10.1007/978-3-642-13675-7_5)
50. 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>
51. Valmari, A.: Stubborn set methods for process algebras. In: DIMACS POMIV, POMIV 1996, pp. 213–231. AMS Press, Inc., New York (1997)
52. Valmari, A., Hansen, H.: Stubborn set intuition explained. In: Petri Nets and Software Engineering 2016, CEUR-WS, pp. 213–232. CEUR (2016)
53. Valmari, A., Vogler, W.: Fair testing and stubborn sets. In: Bošnački, D., Wijs, A. (eds.) SPIN 2016. LNCS, vol. 9641, pp. 225–243. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-32582-8\\_16](https://doi.org/10.1007/978-3-319-32582-8_16)
54. van der Berg, F., Laarman, A.: SpinS: extending LTSmin with Promela through SpinJa. ENTCS **296**, 95–105 (2013)
55. Varpaaniemi, K.: Finding small stubborn sets automatically. In: ISCIS, vol. I, pp. 133–142. Middle East Technical University, Ankara, Turkey (1996)
56. Varpaaniemi, K.: On the stubborn set method in reduced state space generation. Ph.D. thesis, Helsinki University of Technology (1998)
57. Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole partial order reduction. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 382–396. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-78800-3\\_29](https://doi.org/10.1007/978-3-540-78800-3_29)



# Certified Foata Normalization for Generalized Traces

Hendrik Maarand<sup>1(✉)</sup> and Tarmo Uustalu<sup>2,1</sup>

<sup>1</sup> Department of Software Science, Tallinn University of Technology,  
Akadeemia tee 21B, 12618 Tallinn, Estonia

hendrik@cs.ioc.ee

<sup>2</sup> School of Computer Science, Reykjavik University,  
Menntavegi 1, 101 Reykjavik, Iceland

tarmo@ru.is

**Abstract.** Mazurkiewicz traces are a well-known model of concurrency with a notion of equivalence for interleaving executions. Interleaving executions of a concurrent system are represented as strings over an alphabet equipped with an independence relation, and two strings are taken to be equivalent if they can be transformed into each other by repeatedly commuting independent consecutive letters. Analyzing all behaviors of the system can be reduced to analyzing one canonical representative from each equivalence class; normal forms such as the Foata normal form can be used for this purpose. In some applications, it is useful to have commutability of two adjacent letters in a string depend on their left context. We develop Foata normal forms and normalization for Sassone et al.'s context-dependent generalization of traces, formalize this development in the dependently typed programming language Agda and show generalized Foata normalization in action on an example from relaxed shared-memory concurrency (local reads in TSO).

## 1 Introduction

Strings over an alphabet are a simple model of concurrent program behavior presuming that events from different threads are interleaved in an execution of a program. Mazurkiewicz traces [11] are an improvement over strings; a trace corresponds to a set of interleaving executions that can be considered to be equivalent. Traces are equivalence classes of strings. Two strings are taken to be equivalent if they can be transformed to each other by a finite number of commutations of adjacent letters. Commutation is allowed for letters in a given irreflexive and symmetric binary relation, called the independence relation. Traces therefore enable distinguishing concurrent events and causally related events.

While traces are sets of strings, in practice it is desirable to deal with single strings representing these sets canonically. Ideally, such representatives should be strings in some kind of normal form, with normality defined by a decidable predicate. In every trace, there should be exactly one normal form. In particular, given a string, it should be possible to compute its normal form, i.e., the normal

form in its equivalence class. One natural such normal form is the Foata normal form, corresponding to maximally parallel executions; the Foata normal form is well known and understood. (Another such normal form is the lexicographic normal form; yet another is dependence graphs.) The possibility to reduce exploring the full set of executions of a program to exploring the executions in normal form is important in practice. It is often referred to as partial-order reduction.

In some concurrency applications, one needs to depart from standard trace theory by making commutability of two adjacent letters in a string depend on their position in it, specifically their left context. The idea is that this context or history functions as a kind of state, affecting commutability. For this generalization, the independence relation is made dependent on a string parameter for the context. To behave reasonably, it has to meet some well-behavedness conditions. Above all, it must be consistent, i.e., stable under equivalence of contexts, but usually more is required. The exact necessary conditions depend on the application at hand; different sets of conditions have been considered by different authors.

It is natural to ask whether Foata normalization can work also for generalized traces. In this paper, we study and answer this question. On the first look, the prospects for a positive answer are unclear, as the situation is subtler than for standard traces. It is not immediate that the concept of a Foata normal form is reasonable at all—the order of letters in a step should not matter, but their contexts depend on it—or that the normalization function can be defined as in the standard case as one traversal of the given string—a priori, independence or dependence between letters in a string might not remain invariant under inserting an additional letter. But, as it turns out, everything works out well, if one assumes consistency and the coherence conditions introduced by Sassone et al. [15]. Still the definitions and proofs require considerably more care than in the standard case. Especially the proofs become quite subtle, it is easy to make mistakes. This makes context-dependent Foata normalization a good exercise in certified programs and proofs. We conducted this exercise in the dependently typed programming language Agda [12].

The contribution of this paper thus consists in developing the theory of generalized Foata normalization and formalizing it. We also demonstrate the usefulness of this generalization. The reporting is organized as follows. We first introduce both standard Mazurkiewicz traces and Foata normal forms and normalization as well as the context-dependent generalization using mathematical notation on a high level in Sect. 2. Then, in Sect. 3, we describe the formalization of the generalization in Agda, disclosing a fair degree of detail of not only the definitions, but also the proofs. In Sect. 4, we demonstrate that context-dependent independence arises naturally in relaxed shared-memory concurrency. Then we briefly discuss related work and conclude.

Our Agda formalization is at <http://cs.ioc.ee/~hendrik/code/generalized-traces/agda.zip>. The code works with Agda version 2.5.2 and Agda standard library version 0.13. It consists of approx. 3800 lines of code whereof standard traces take approx. 1100 lines, generalized traces approx. 1300 lines and the rest



is utility code for cons/snoc list manipulation and similar purposes. The formalization of generalized traces does not depend on the formalization of standard traces.

We assume no knowledge of trace theory from the reader, introducing all relevant concepts and facts. When describing our formal development, we show snippets of Agda code, but we also comment them.

## 2 Traces, Foata Normal Forms and Normalization

Traces are equivalence classes of strings with respect to a congruence relation that allows to commute certain pairs of letters. More precisely, an *alphabet*  $\Sigma$  is a (non-empty) set whose elements we call letters; letters model events. A *string* is a list of letters, i.e., an element of  $\Sigma^*$ , the free monoid on  $\Sigma$ . Strings are used to model executions of programs. An *independence relation*  $I \subseteq \Sigma \times \Sigma$  is an irreflexive and symmetric binary relation. Its complement  $D = (\Sigma \times \Sigma) \setminus I$ , which is reflexive and symmetric, is called the *dependence relation*. Intuitively, if  $aIb$ , then the strings  $uabv$  and  $ubav$  represent the “same” execution. We define  $\sim \subseteq \Sigma^* \times \Sigma^*$  to be the least relation such that  $aIb$  implies  $uabv \sim ubav$  and define (Mazurkiewicz) *equivalence*  $\sim^*$  to be its reflexive-transitive closure. A (Mazurkiewicz) *trace* is an equivalence class of strings wrt.  $\sim^*$ , i.e., an element of the quotient set  $\Sigma^*/\sim^*$ , which is the free partially commutative monoid.

For example, if  $\Sigma = \{a, b, c, d\}$  and  $I$  is the least symmetric relation satisfying  $aIb, aId, bId, cId$ , then the strings  $abcd$  and  $bdac$  are equivalent, since  $abcd \sim bacd \sim badc \sim bdac$ , but  $acbd$  is not equivalent to them. The strings  $abcd, abdc, adbc, bacd, badc, bdac, dabc, dbac$  form one equivalence class of strings or trace. Another is  $\{acbd, acdb, adcb, dacb\}$ . Altogether, there are only four traces containing each letter of  $\Sigma$  exactly once.

A (Foata) *step* is a non-empty set  $s$  of pairwise independent letters, i.e., for any different  $a, b \in s$ , we require  $aIb$ . If we are given a strict total order, i.e., a transitive and asymmetric relation,  $\prec \subseteq \Sigma \times \Sigma$  on letters, then we can equivalently define that a step is a  $\prec$ -sorted non-empty list of pairwise independent letters. A (Foata) *normal form*  $n : \text{Nf}$  is a list  $s_0 \dots s_{m-1}$  of steps such that, for any  $i < m$ , unless  $i = 0$ , for any  $b \in s_i$ , there is  $a \in s_{i-1}$  such that  $aDb$ .

To continue the example above, suppose that  $\prec$  is given by  $a \prec b \prec c \prec d$ . We then have that  $(abd)(c)$  is a normal form, since  $a, b, d$  are pairwise independent and  $c$  is dependent with, for instance,  $b$ . However,  $(a)(c)(bd)$  is not a normal form: we have  $bId, aDc, cDb$ , but  $cDd$  does not hold.

Viewing steps as non-empty lists, we have a straightforward embedding  $\text{emb} : \text{Nf} \rightarrow \Sigma^*$  of normal forms into strings given by concatenation.

Assuming  $I$  and  $\prec$  to be decidable, every trace has a unique normal form, i.e., we have a function  $\text{norm} : \Sigma^* \rightarrow \text{Nf}$  such that  $u \sim^* \text{emb}(\text{norm } u)$  (existence of a normal form),  $u \sim^* v$  implies  $\text{norm } u = \text{norm } v$  (soundness of norm) and  $n = \text{norm}(\text{emb } n)$  (stability of norm). Existence straightforwardly gives that  $\text{norm } u = \text{norm } v$  implies  $u \sim^* v$  (completeness of norm). From soundness and stability, it follows that  $u \sim^* \text{emb } n$  implies  $\text{norm } u = n$  (uniqueness of a normal form).

The function *norm* is defined quite naturally. It takes a string  $u$  and traverses it from the left to the right, maintaining the normal form of the prefix already seen. If the normal form of this prefix is  $s_0 \dots s_{m-1}$ , then the function finds the greatest  $i \leq m$  such that the next letter  $a$  is dependent with some letter in  $s_{i-1}$  unless  $i = 0$ , and inserts  $a$  into step  $s_i$ , if  $i < m$  (commuting past all steps whose all letters it is independent with), or adds a new singleton step  $s_m$  consisting initially of  $a$  only, if  $i = m$ . Intuitively, the given string is thus rearranged into a maximally parallel form.

In our example, the string  $acbd$  is normalized as follows. We first make a normal form consisting of a single step ( $a$ ). Letter  $c$  is dependent with  $a$ , thus cannot be added to this step, so we start a new step:  $(a)(c)$ . Letter  $b$  is dependent with  $c$ , so we start a new step again:  $(a)(c)(b)$ . Letter  $d$  is independent with all of  $b, c, a$ , so we insert it into the first step:  $(ad)(c)(b)$ .

In generalized traces, the commutability of two adjacent letters depends on their left context, corresponding to the execution so far. The independence relation is parameterized by a string for this context. More precisely, independence is an assignment of an irreflexive and symmetric relation  $I_u \subseteq \Sigma \times \Sigma$  to every string  $u$ . This family of relations must be *consistent*, i.e., stable under equivalence in the sense that  $u \sim^* v$  and  $aI_u b$  imply  $aI_v b$ . Sassone et al. [15] require also the following *coherence* conditions:

1.  $aI_u b$  and  $bI_{ua} c$  and  $aI_{ub} c$  imply  $aI_u c$ ,
2.  $aI_u b$  and  $bI_u c$  imply ( $aI_u c$  iff  $aI_{ub} c$ ).

Consistency is a very basic hygiene condition. The coherence conditions are more difficult to make sense of and memorize. They also have many similar-looking consequences. One way to see the coherence conditions is to say that they are the smallest set of conditions guaranteeing that any choice of three conditions, one from each of the following three pairs, implies the other three conditions:  $(aI_u b, aI_{ub} c)$ ,  $(bI_u c, bI_{ua} c)$ ,  $(aI_u c, aI_{ub} c)$ . This is with the exception of the choice of the second condition from each pair; from these three conditions one cannot conclude anything. For example,  $aI_u b$  and  $bI_{ua} c$  and  $aI_{ub} c$  imply not only  $aI_u c$  and  $bI_u c$  (both by 1.), but also  $aI_{uc} b$  (follows from those by 2. $\Rightarrow$ ).

To illustrate generalized traces, let us modify our example. We take  $I$  now to be the least consistent, coherent family of symmetric relations such that  $aI_{\square} b$ ,  $aI_{\square} d$ ,  $bI_a d$ ,  $bI_{ac} d$ ,  $cI_{ab} d$ . (We write  $\square$  for the empty string.) Explicitly, this means that we also have  $bI_{\square} d$  (by 2. $\Leftarrow$ ),  $aI_a b$ ,  $aI_b d$  (by 2. $\Rightarrow$ ) and  $cI_{ba} d$  (by consistency). Now  $abcd$  has the same equivalence class as before, but  $acbd$  is only equivalent to  $acdb$ , leaving  $adcb$  and  $dacb$  in a different equivalence class.

We would like to scale Foata normalization to generalized traces. Our adjustment of the definition of a Foata normal form is as follows. A (*Foata*) *normal form* is a list  $s_0 \dots s_{m-1}$  of non-empty lists of letters (*steps*) such that, for all  $i < m$ ,

- for any  $a, b \in s_i$ , if  $a \neq b$ , then  $aI_{s_0 \dots s_{i-1}} b$ ,
- unless  $i = 0$ , for any  $b \in s_i$ , there is  $a \in s_{i-1}$  such that  $aD_{s_0 \dots s_{i-2}} b$ ,
- $s_i$  is  $\leftarrow$ -sorted.

Note that in the above conditions dependence or independence is stated wrt. contexts of whole steps rather than contexts of individual letters in a step. This is motivated by the intuition that the letters in a step should be concurrent and their order of appearance in the step should be incidental (depending on the chosen total strict order on the alphabet, which should be immaterial).

We thus have a sensible-looking definition, but does it work? And can the normalization function be defined in the same way as for standard traces? In the next section, we will show this to be the case, by describing our formalized development that includes proofs. The coherence conditions turn out to be instrumental in ensuring that we are indeed entitled to check coherence for contexts of steps in normal forms rather than contexts of individual letters.

In our example, the equivalence class of  $abcd$  consists of 8 strings, with  $(abd)(c)$  the normal form, as before. The equivalence class of  $acbd$  is  $\{acbd, acdb\}$ , with  $(a)(c)(bd)$  the normal form. The equivalence class of  $adcb$  is  $\{adcb, dacb\}$ , with  $(ad)(c)(b)$  the normal form.

The significance of coherence can be demonstrated already on this small example. If we had  $bD_{\square}d$  instead of  $bI_{\square}d$  (violating 2. $\Leftarrow$ ), then  $(abd)(c)$  would cease to be a normal form under our chosen definition of normal forms. Instead, both  $(ab)(cd)$  and  $(ad)(b)(c)$  would be normal forms, although  $abcd \sim abdc \sim adbc$ , so normal forms would not be unique. (Our chosen normalization function would return  $(ab)(cd)$  for  $abcd$  and  $abdc$ ,  $(ad)(b)(c)$  for  $adbc$ .) If we had  $aD_{\square}b$  and  $aD_{\square}d$  instead of  $aI_{\square}b$  and  $aI_{\square}d$  (violating 2. $\Rightarrow$ ), then the strings  $bdac$  and  $dbac$  would only be equivalent to each other and without a normal form under our chosen definition of normal forms. (Our chosen normalization function would return  $(abd)(c)$  for  $bdac$  and  $dbac$ , which is not a normal form of these strings.)

### 3 Formalization

We will now present an overview of our Agda formalization of generalized traces, Foata normal forms and normalization. Showing Agda code, we will use some unofficial shortcuts to enhance readability; above all, we will typically omit implicit arguments in type signatures. While we will describe the definition of the normalization function in detail, for lemmas of the correctness proof, we will only give the type (the statement) and omit the proof.

#### 3.1 Traces

We start our formalization from an alphabet  $A$  that we assume to be given, therefore we have defined it as a module parameter in Agda.

```
A : Set
```

The next component is that of a string (or word) over the alphabet  $A$ .

```
String = List A
String> = List> A
```

We work with two versions of strings—cons-lists and snoc-lists of letters—since, in some situations, we prefer to access the letters from the left end and in some situations from the right end. In fact, we will see that our normalization function manipulates a zipper: it traverses the input string (a cons-list) from the left and inserts every letter into the accumulated normal form, which is a snoc-list (of steps). `List>` is the type of snoc-lists. We have marked the usual list operations on snoc-lists with a trailing `>` to emphasize that they are for lists where the head element is on the right. Conversions between cons-lists and snoc-lists are denoted `c2s` and `s2c`.

We also assume a context-dependent independence relation on `A` that is both irreflexive and symmetric.

```
_I[_]_ : A → String> → A → Set
I-irr  : ∀ c → Irreflexive _I[ c ]_
I-sym  : ∀ c → Symmetric  _I[ c ]_
```

The underscores in mixfix operator identifiers mark the places where the arguments will go, in the order given in the type signature. `a I[ c ] b` means that, in the context `c` (which is a `String>`), the letters `a` and `b` are independent. We model the context as a snoc-list since most of the time we need to access the right end of the context. We define the context-dependent dependence relation `a D[ c ] b` as the negation (complement) of independence.

We say that two strings are one-step convertible if they differ only by the ordering of a pair of adjacent independent letters.

```
data _~[_]_ : String → String> → String → Set where
  swap : a I[ c ++> c2s u ] b → (u ++ a :: b :: v) ~[ c ] (u ++ b :: a :: v)
```

Note that the letters `a` and `b` are independent in the context `c` plus `u`, which is the common prefix of the two strings. We now take the reflexive-transitive closure of this one-step convertibility relation.

```
data _~[_]*_ : String → String> → String → Set where
  refl*      : u ≅ v → u ~[ c ]* v
  swap-trans* : u ~[ c ] t → t ~[ c ]* v → u ~[ c ]* v
```

Note that the context `c` is fixed in the case of `swap-trans*`, which means that, in `u ~[ c ]* v`, the strings that are actually considered equivalent are `c` plus `u` and `c` plus `v`, but the context `c` stays the same and no exchanges can be done in the context.

We are often working with strings in the empty context (we are looking at whole strings). For this case, we define the following abbreviations.

```
u ~ v = u ~[ [] ] v
u ~* v = u ~[ [] ]* v
```

### 3.2 Normal Forms

We represent a Foata normal form as a snoc-list of steps and a step as a snoc-list of letters.

```
Step = List> A
Foata = List> Step
```

These are the datatypes for “raw” steps and normal forms, we have not yet imposed the relevant well-formedness conditions. The embedding function is defined by flattening the two-layer snoc-list and converting the result to a cons-list.

```
emb : Foata → String
emb ss = s2c (concat> ss)
```

To define the well-formedness predicate for `Step`, we assume that our given alphabet  $A$  has a strict total order  $\_<\_$  defined on it. We also lift the independence relation from a binary relation on letters to a relation between a step and a letter.

```
_<_ : A → A → Set
sto< : StrictTotalOrder _<_

_■I[_]_ : Step → String> → A → Set
s ■I[ c ] a = All> (λ b → b I[ c ] a) s
```

$All> P\ xs$  means that the predicate  $P$  holds on every element of the snoc-list  $xs$ .

```
data StepOk : String> → Step → Set where
  sngl : (a : A) → StepOk c [ a ]>
  snoc : StepOk c (s ∷ a') → (a : A) → a' < a → (s ∷ a') ■I[ c ] a
    → StepOk c (s ∷ a' ∷ a)
```

A well-formed step is either a singleton (in a context  $c$ ) or it consists of a well-formed step to which a new letter is added on the right, which has to be greater than the previous rightmost letter. Also, the old step and the new letter must be independent.

A small remark here is that `StepOk c s` is not necessarily a proposition (there can be more than one inhabitant of this type). If we were in a situation where we have  $p, q : StepOk\ c\ s$  and we had to show  $p \cong q$ , we would have two options. Either we would have to assume that  $a < b$  and  $a\ I[ c ]\ b$  are propositions (for any  $a, b$  and  $c$ ) or we would have to change the definition of `StepOk` to use propositional truncation when using  $\_<\_$  and  $\_I[_]_\_$  (to have a normalization function, we must assume that both of these predicates are decidable, and this gives us effective truncation as a byproduct).

To define the well-formedness predicate for `Foata`, we first lift the dependence relation to a relation between a step and a letter (to describe when a letter is “supported” by a step). We also extend this to normal forms.

```
_◆D[_]_ : Step → String> → A → Set
s ◆D[ c ] a = Any> (λ b → b D[ c ] a) s
```

```

_◆D'_ : Foata → A → Set
[]      ◆D' a = ⊤
(ss ::> s) ◆D' a = s ◆D' [ concat> ss ] a

```

$\text{Any} > P \text{ xs}$  is the type of existence of an element  $x$  in  $\text{xs}$  such that  $P \ x$  holds.  $\top$  is the unit type (the trivially true proposition). Note the context used in the non-empty case of  $\text{◆D}'$ .

```

data FoataOk : Foata → Set where
  empty : FoataOk []
  step  : FoataOk ss → StepOk (concat> ss) s → All> (λ a → ss ◆D' a) s
        → FoataOk (ss ::> s)

```

A well-formed normal form can either be empty or it can consist of a well-formed normal form to which a step is added that must be well-formed in the context of this normal form. An additional condition for the result to be well-formed is that every letter in the new step is supported by the preceding normal form.

Similarly to  $\text{StepOk}$ ,  $\text{FoataOk}$  is not a proposition. Even if we redefine  $\text{StepOk}$  so that it becomes a proposition, we still have the proof that the letters of the new step are supported by the normal form in the  $\text{step}$  case. If there are multiple candidates to support a letter in the new step, then we should give a canonical way to pick one of them. For example, we could always pick the rightmost dependent letter from the previous step as the support. Another option is again to use propositional truncation.

### 3.3 Normalization

We define a function  $\text{find}>$ , which given a decidable predicate and a list, splits the input list into two lists so that all of the elements in the second list (on the right) satisfy the predicate and the rightmost element in the first list violates the predicate, or the first list is empty. Note that the decidable predicate  $P$  here is context-dependent (it looks at both the head and tail of the  $\text{snoc}$ -list).

```

find> : (∀ xs x → Dec (P xs x)) → List> X → List> X × List> X
find> d? [] = [], []
find> d? (xs ::> x) with d? xs x
find> d? (xs ::> x) | yes p = let ys , zs = find> d? xs in ys , zs ::> x
find> d? (xs ::> x) | no ¬p = xs ::> x , []

```

If we have a step and a letter that should be added to that step, then we can accomplish this by finding the right place for the new letter according to the ordering  $\text{<?}$ . We assume decidability of  $\text{<?}$ .

```

_<?_ : (a b : A) → Dec (a < b)

push : Step → A → Step
push s a = let ls , rs = find> (λ _ b → a <? b) s in ls ::> a ++ rs

```

The properties of `find>` give us that all letters in `rs` are greater than `a` and the rightmost letter in `ls` is less than `a` or `ls` is empty. If `s` happened to be a well-formed step, then so are also both `ls` and `rs`. Given a normal form and a letter, we need to figure out the correct step for that letter and push it into that step. We assume decidability of `_I[_]_`.

```

_I[_]?_ : (a : A) → (c : String) → (b : A) → Dec (a I[ c ] b)

insert : Foata → A → Foata
insert ss a with find> (λ xs x → x ■I[ concat> xs ]? a) ss
insert ss a | ls , []      = ls => [ a ]>
insert ss a | ls , rs => r = let s , rs' = first rs r in ls => push s a ++> rs'
    
```

Here, we use `find>` to split the normal form `ss` into two parts `ls` and `rs` so that `a` and all of the steps in `rs` are independent and `a` and the last step in `ls` are dependent or `ls` is empty. Note that while `a` and the steps in `rs` are independent, the contexts for those steps (and independence relations) are different. We pattern match on `rs` to decide whether to add a new step or not. If `rs` is empty, then `ss` already supports `a` and we must add a new step, otherwise, we push `a` to the leftmost step in `rs`. Here, `■I[_]?_` lifts `_I[_]?_` from deciding independence of letters to deciding independence of a step and a letter.

With `insert` in place, we can now define a normalization function that traverses the string from the left to the right and inserts each letter into the correct position in the accumulated normal form.

```

norm' : Foata → String → Foata
norm' ss []      = ss
norm' ss (a :: t) = norm' (insert ss a) t

norm : String → Foata
norm t = norm' [] t
    
```

This is our normalization function, but it produces “raw” normal forms. We will now show that the functions we defined are “good” in the sense that they produce good output from good input. We begin with `push`.

```

push0k : Step0k c s → (a : A) → s ■I[ c ] a → Step0k c (push s a)
    
```

Given that `s` is a well-formed step and `s` and `a` are independent, the result of `push s a` is also a well-formed step. To construct `Step0k c (push s a)`, we need to show that the letters in `push s a` are ordered and independent. They are ordered since the letters in `s` are ordered and `push` uses `find>` to find a position in the list which respects the ordering. The letters are independent since the letters in `s` are independent and `a` and `s` are independent, which means that `a` and any subset of `s` (including the results of `find>`) are independent. The context `c` stays fixed inside a step.

We now continue with `insert` and show that it produces a well-formed normal form when given a well-formed normal form and a letter.

```

insert0k : Foata0k ss → (a : A) → Foata0k (insert ss a)
    
```

Compared to push0k, things get much more involved here. Namely, when we insert a letter into a normal form, we are modifying a step somewhere in the middle of the normal form. As a result, the contexts for the invariants of the steps to the right of the modified step have changed.

For example, if we have a normal form  $stuv$  consisting of the steps  $s$ ,  $t$ ,  $u$  and  $v$ , then inserting  $a$  into  $stuv$  that will go into the step  $t$  changes the following. The letters in  $u$  must now be independent in the context  $s(\text{push } t \ a)$  instead of  $st$  and the letters in  $v$  must now be independent in the context  $s(\text{push } t \ a)u$ . Also, every letter of  $v$  must now have support from a letter in  $u$  in the context  $s(\text{push } t \ a)$  instead of  $st$ . One consequence of the consistency and coherence axioms is that when we do an insert, then we do not need to renormalize the part of the normal form for which the context changed, we can prove that the invariants still hold.

$$\begin{aligned} \text{I-cons} & : c \sim^* c' \rightarrow a \text{ I}[ c2s \ c ] \ b \rightarrow a \text{ I}[ c2s \ c' ] \ b \\ \text{I-co1} & : a \text{ I}[ c ] \ b \rightarrow a \text{ I}[ c \Rightarrow b ] \ d \rightarrow d \text{ I}[ c \Rightarrow a ] \ b \rightarrow a \text{ I}[ c ] \ d \\ \text{I-co2-e} & : a \text{ I}[ c ] \ b \rightarrow b \text{ I}[ c ] \ d \rightarrow a \text{ I}[ c ] \ d \quad \rightarrow a \text{ I}[ c \Rightarrow b ] \ d \\ \text{I-co2-r} & : a \text{ I}[ c ] \ b \rightarrow b \text{ I}[ c ] \ d \rightarrow a \text{ I}[ c \Rightarrow b ] \ d \rightarrow a \text{ I}[ c ] \ d \end{aligned}$$

During an insert, we need to make sure that every step that we overtake with the new letter still satisfies the invariants, pairwise independence and support.

For pairwise independence, we need to repeatedly apply the I-co2-e axiom. To move a letter  $b$  past a step  $s$ , we must have  $s \blacksquare \text{I}[ c ] \ b$ . Since  $s$  is a (valid) step, it must be that the letters in  $s$  are pairwise independent. This means that, for every  $a$  and  $d$  in  $s$ , we have that  $a \text{ I}[ c ] \ b$ ,  $d \text{ I}[ c ] \ b$  and  $a \text{ I}[ c ] \ d$ , which gives  $a \text{ I}[ c \Rightarrow b ] \ d$ . Consequently, the letters in  $s$  are still pairwise independent after extending the context with  $b$ .

For support, we think of the situation where we have a context (normal form)  $c$  and steps  $s$  and  $s'$  such that, for every  $d$  from  $s'$ , we have an  $a$  from  $s$  so that  $a \text{ D}[ c ] \ d$ . This stops  $d$  from being moved into the earlier step  $s$ . To move  $b$  past  $s$  and  $s'$  into the context  $c$ , it must be that  $s \blacksquare \text{I}[ c ] \ b$  and  $s' \blacksquare \text{I}[ c \text{ ++} \ s ] \ b$ , which also gives that  $a \text{ I}[ c ] \ b$  and  $d \text{ I}[ c \text{ ++} \ s ] \ b$ . This starts to resemble the contraposition of I-co1:

$$\text{D-co1} : a \text{ I}[ c ] \ b \rightarrow a \text{ D}[ c ] \ d \rightarrow d \text{ I}[ c \Rightarrow a ] \ b \rightarrow a \text{ D}[ c \Rightarrow b ] \ d$$

What we are missing from D-co1 is  $d \text{ I}[ c \Rightarrow a ] \ b$ , but we have  $d \text{ I}[ c \text{ ++} \ s ] \ b$  and we also know that  $a$  is from  $s$ . We can prove the following lemma about extending the context of support:

$$\begin{aligned} \text{PW} & : (X \rightarrow X \rightarrow \text{Set}) \rightarrow \text{List} \times X \rightarrow \text{Set} \\ \text{PW P} \ [ ] & = \top \\ \text{PW P} \ (xs \Rightarrow x) & = \text{PW P } xs \times \text{All} \times (\lambda x' \rightarrow \text{P } x' \ x) \ xs \\ \blacklozenge \text{D-ext-lem} & : d \text{ I}[ c \text{ ++} \ s ] \ b \rightarrow \text{PW } \_ \text{I}[ c ] \_ \ (s \Rightarrow b) \\ & \rightarrow s \blacklozenge \text{D}[ c ] \ d \rightarrow s \blacksquare \text{I}[ c \Rightarrow b ] \ d \rightarrow \perp \end{aligned}$$

This expresses the fact that, under suitable conditions, it cannot be that  $d$  is supported by (the step)  $s$  in the context  $c$  but is not supported by  $s$  in the extended context  $c \Rightarrow b$ . The first condition is that the letters  $b$  (that we add



to the context) and  $d$  are independent (in the context  $c$  plus  $s$ ). We also require that  $s$  can be viewed as a step (its letters are pairwise independent) and that the letter  $b$  and the step  $s$  are independent. The last two conditions are expressed by  $\text{PW\_I}[c]_-(s \Rightarrow b)$ , which says that the predicate  $\_I[c]_-$  holds pairwise between the letters in  $s \Rightarrow b$ .

$s \blacklozenge I[c] d$  and  $s \blacksquare I[c \Rightarrow b] d$  give us that there is a letter  $a$  in  $s$  such that  $a \blacklozenge I[c] d$  and  $a \blacksquare I[c \Rightarrow b] d$ . We can apply contraposition of  $I\text{-co2-r}$  to get that both of  $a \blacksquare I[c] b$  and  $d \blacksquare I[c] b$  cannot hold. From  $\text{PW\_I}[c]_-(s \Rightarrow b)$ , we get  $a \blacksquare I[c] b$ , which means that  $d \blacklozenge I[c] b$  must hold. We derive a version of  $I\text{-co1}$  where one of the letters has been replaced with a step.

$$\blacksquare I\text{-co1} : s \blacksquare I[c] b \rightarrow d \blacksquare I[c \Rightarrow s] b \rightarrow s \blacksquare I[c \Rightarrow b] d \rightarrow \text{PW\_I}[c]_-(s \Rightarrow b) \rightarrow d \blacksquare I[c] b$$

All of the arguments of this rule match the  $I$ -arguments of  $\blacklozenge\text{-ext-lem}$ , so we can derive  $d \blacksquare I[c] b$ . This cannot be since we previously derived  $d \blacklozenge I[c] b$ .

$\blacklozenge\text{-ext-lem}$  allows us to extend the context with a suitable letter, but we also need to “slide” the new letter to the position where  $\text{insert}$  would take it.

$$\text{slide-step} : \text{PW\_I}[c]_-(s \Rightarrow b) \rightarrow s2c(c \Rightarrow s \Rightarrow b) \sim^* s2c(c \Rightarrow b \Rightarrow s)$$

This says that, if we have a letter  $b$  and step  $s$  that are independent, then we can slide  $b$  past  $s$  and the resulting string is equivalent to the one we started with. By repeatedly applying  $\text{slide-step}$ , we can move a letter past multiple steps and still preserve equivalence. This allows us to prove that  $\text{insert}$  is good. It follows that  $\text{norm}$  is also good:

$$\begin{aligned} \text{normOk}' & : \text{FoataOk } ss \rightarrow (t : \text{String}) \rightarrow \text{FoataOk } (\text{norm}' ss t) \\ \text{normOk} & : (t : \text{String}) \rightarrow \text{FoataOk } (\text{norm } t) \end{aligned}$$

### 3.4 Properties

We will now proceed to soundness and completeness of our normalization algorithm. The first lemma is about  $\text{insert}$  and it says that inserting a letter into a normal form and then embedding into a string is equivalent to first embedding the normal form into a string and adding the letter to the end.

$$\text{insert-lem} : \text{FoataOk } ss \rightarrow (a : A) \rightarrow \text{emb } (\text{insert } ss a) \sim^* (\text{emb } ss ++ [a])$$

This lemma is essentially  $\text{slide-step}$  lifted to the case of multiple steps and defined in terms of  $\text{insert}$ . This gives us the necessary tools to prove the existence of normal forms, which will then lead to completeness.

$$\begin{aligned} \text{nf-exists}' & : \text{FoataOk } ss \rightarrow (t : \text{String}) \rightarrow \text{emb } (\text{norm}' ss t) \sim^* (\text{emb } ss ++ t) \\ \text{nf-exists} & : (t : \text{String}) \rightarrow \text{emb } (\text{norm } t) \sim^* t \end{aligned}$$

$$\text{completeness} : \text{norm } u \cong \text{norm } v \rightarrow u \sim^* v$$

To prove soundness, we first show that our normalization operation commutes for independent letters. We start with  $\text{push}$ .

$$\begin{aligned} \text{push-commutes} &: \text{Step0k } c \ s \rightarrow a \ I[ \ c \ ++> \ s \ ] \ b \rightarrow s \ \blacksquare I[ \ c \ ] \ a \rightarrow s \ \blacksquare I[ \ c \ ] \ b \\ &\rightarrow \text{push } (\text{push } s \ a) \ b \cong \text{push } (\text{push } s \ b) \ a \end{aligned}$$

This says that, if we have a well-formed step  $s$  and two letters  $a$  and  $b$  that are independent and that fit into  $s$  ( $a$ ,  $b$  and  $s$  are independent), then it does not matter in which order we push them into the step. Since we have a strict total order on the alphabet and the letters in a valid step must be ordered, there is only one way to put  $a$  and  $b$  into  $s$  such that the result is ordered. We continue with a similar lemma about `insert`.

$$\begin{aligned} \text{insert-commutes} &: \text{Foata0k } ss \rightarrow a \ I[ \ \text{concat}> \ ss \ ] \ b \\ &\rightarrow \text{insert } (\text{insert } ss \ a) \ b \cong \text{insert } (\text{insert } ss \ b) \ a \end{aligned}$$

This says that, if we have a normal form  $ss$  and two letters that are independent in the context of that normal form, then it does not matter in which order we insert them. We prove this by case analysis on the steps that  $a$  and  $b$  go to.

If both of these letters are supported by  $ss$ , then we must add a new step and show that it does not matter whether we add a new step by a singleton  $a$  or  $b$ . It may be that one of the letters fits into the existing steps and the other does not, in which case we have to show that, even if we add a new step, the other letter will still go past it (as the letters are independent) and that we need to add a new step even if we insert the other letter before. The last option is that both of them fit into the existing steps. If both of them go to the same step, then we apply `push-commutes`. Otherwise, we need to show that a letter goes to the same step whether we insert the other letter or not. We make use of some lemmas to consider an `insert` operation as a push. For example:

$$\begin{aligned} \text{insert-last} &: ss \ \blacklozenge D' \ a \rightarrow s \ \blacksquare I[ \ \text{concat}> \ ss \ ] \ a \\ &\rightarrow \text{insert } (ss \ \Rightarrow s) \ a \cong ss \ \Rightarrow \text{push } s \ a \end{aligned}$$

Here we have that  $a$  is supported by  $ss$ , which means that `insert` cannot take this letter any further. We also have that  $a$  can be pushed into  $s$  since they are independent. This gives us that, in this situation, `insert (ss  $\Rightarrow$  s) a` is the same as `ss  $\Rightarrow$  push s a`. These tools allow us to prove that `insert` commutes. The fact that `norm'` commutes follows immediately.

$$\begin{aligned} \text{norm'-commutes} &: \text{Foata0k } ss \rightarrow a \ I[ \ \text{concat}> \ ss \ ] \ b \\ &\rightarrow \text{norm}' \ ss \ (a \ :: \ b \ :: \ []) \cong \text{norm}' \ ss \ (b \ :: \ a \ :: \ []) \end{aligned}$$

We are now ready to prove soundness of the normalization algorithm. We first show that `norm'` is sound for strings that are one-step convertible.

$$\text{sound}\sim : \text{Foata0k } ss \rightarrow t \ \sim[ \ \text{concat}> \ ss \ ] \ t' \rightarrow \text{norm}' \ ss \ t \cong \text{norm}' \ ss \ t'$$

It is important to note that  $t$  and  $t'$  are convertible in the possibly non-empty context `concat> ss`. The proof basically splits  $t$  and  $t'$  into pieces, so that we can focus on the pair of letters that make the two strings different and apply `norm'-commutes`. A useful lemma at this point is `norm'-append` that exposes the compositional nature of our normalization algorithm.

$$\text{norm'-append} : (t \ t' : \text{String}) \rightarrow \text{norm}' \ ss \ (t \ ++ \ t') \cong \text{norm}' \ (\text{norm}' \ ss \ t) \ t'$$

We lift  $\text{sound} \sim$  to its reflexive-transitive closure. This also gives that  $\text{norm}$  is sound.

$\text{sound}^* : \text{Foata}0k \text{ ss} \rightarrow t \sim [\text{concat} > \text{ss}]^* t' \rightarrow \text{norm}' \text{ ss } t \cong \text{norm}' \text{ ss } t'$   
 $\text{soundness} : t \sim^* t' \rightarrow \text{norm } t \cong \text{norm } t'$

We can now decide equivalence of two strings by first normalizing the strings and then checking if the normal forms are equal. If they are, then  $\text{completeness}$  says that the strings are equivalent, and, if the normal forms are not equal, then  $\text{soundness}$  says that the strings cannot be equivalent. To do this, we assume decidable equality on the alphabet.

$\_ \cong ? \_ : (a \ b : A) \rightarrow \text{Dec } (a \cong b)$   
 $\text{equivalent?} : (u \ v : \text{String}) \rightarrow \text{Dec } (u \sim^* v)$

We also have that normal forms are stable, meaning that, if we (re-)normalize the embedding of a normal form, then we get back the same normal form.

$\text{stability} : \text{Foata}0k \text{ ss} \rightarrow \text{norm } (\text{emb } \text{ss}) \cong \text{ss}$

We also get uniqueness of normal forms, meaning that, if we have two normal forms whose embeddings are equivalent, and thus come from the same equivalence class, then they must be equal.

$\text{nf-unique} : \text{Foata}0k \text{ ss} \rightarrow \text{Foata}0k \text{ ss}' \rightarrow \text{emb } \text{ss} \sim^* \text{emb } \text{ss}' \rightarrow \text{ss} \cong \text{ss}'$

## 4 Example: Local Reads in TSO

Here we will give a small example where generalized traces are needed to describe the behaviour of a concurrent system reasonably precisely. The example comes from shared-variable concurrency in a system with write buffers which corresponds to the Total Store Order (TSO) relaxed memory model from the SPARC hierarchy [16].

The machine that we are going to model consists of processors and shared memory where each processor has local registers and a single write buffer. A program is a list of read and write instructions. The execution of a write instruction generates two events. First, the write is added to the buffer (the main event). Later, it is flushed to memory (the shadow event). The processor sees the memory “through” the buffer: if there are any writes to the variable  $x$  in the buffer, then the processor sees the value of the last write to  $x$  as the value of  $x$ , otherwise, it sees the value currently in memory.

We can represent program executions on this machine as words over the alphabet of events. We can also consider a dependence relation on this alphabet. Two events from different processors are dependent when they access the same memory location and at least one of them is a write. Two events from the same processor are dependent, if they are both main events (we respect the program order) or both shadow events (the buffer is *first-in-first-out*) or they are a corresponding pair of a main and a shadow event (a shadow event cannot happen before its cause, the main event).

Let us consider the following program where two processors write to variable  $x$  and one of them also reads from  $x$ .

$P_1$	$P_2$
$(a, a') [x] := 1$	$(c, c') [x] := 2$
$(b) \quad r1 \quad := [x]$	

Our first scenario considers the execution  $acc'ba'$ , which has the same meaning as  $acbc'a'$ . This is because, in both of these executions,  $b$  happens before  $a'$ , so there is a write in the buffer and  $b$  reads its value from that. Whether  $c'$  happens before or after  $b$ , does not matter. It seems reasonable in this situation to say that  $b$  and  $c'$  are independent.

The second scenario considers the execution  $aca'bc'$ , which has a different meaning than  $aca'c'b$ . This is because  $b$  happens after  $a'$  and thus  $b$  reads its value from memory since the buffer is empty. In the first execution,  $b$  reads the value written by  $a'$ , and, in the other execution, it reads the value written by  $c'$ . Here, we would like to say that  $b$  and  $c'$  are dependent.

If we were to model this program using standard traces, we would have to set  $b$  and  $c'$  to be dependent; otherwise, the two executions of the second scenario would be considered equivalent, but they have different behaviour in the sense of taking the same initial state to different final states. This forced choice, however, distinguishes the two executions of the first scenario that behave the same way. This is an imprecision, we have more equivalence classes than desirable. Generalized traces allow us to say that  $b$  and  $c'$  are dependent only if there are no writes to  $x$  by the first processor in the buffer. This is information that can be read off the given context. The context should not contain  $a$  without also  $a'$  somewhere to its right, so, for example, the empty context and  $aa'$  make  $b$  and  $c'$  dependent, but  $ac$  does not.

## 5 Related Work

Traces were introduced into concurrency theory by Mazurkiewicz [11], but they originate from the enumerative combinatorics work by Cartier and Foata [3]. In particular, Foata normalization is from that work. Foata normalization is described in many of the standard expositions of trace theory, e.g., [1, 5].

Generalizing traces for context-dependent independence has been considered by several authors, but with different well-behavedness conditions on independence. Sassone et al. [15] introduced context-dependent independence as we have considered it in this paper. Katz and Peled [9] introduced state-dependent independence, considering a coherence condition that in our setting would amount to  $aI_{ub}$  and  $bI_{ua}c$  implying  $(aI_{uc}$  iff  $aI_{ub}c)$ . This condition is equivalent to the conjunction of conditions 1. and  $2.\Rightarrow$  of Sassone et al. Droste [6], in a work on concurrent automata, again with state-dependent independence, required what would in our setting amount to  $aI_{ub}$ ,  $bI_{uc}$ ,  $aI_{ub}c$  implying  $aI_{uc}b$ ,  $bI_{ua}c$ ,  $aI_{uc}$ , i.e., condition  $2.\Leftarrow$  and a little more. Hoogers et al. [8] developed local traces where independence relates lists of steps to steps. This is a different setup where

coherence conditions like those of Sassone et al. do not arise, because one only works with contexts of steps, not contexts of individual letters.

Partial-order reduction and use of representatives in model-checking, originally proposed by Godefroid [7] and Peled [14], are in wide use. Dynamic partial-order reduction for stateless model-checking of relaxed memory concurrent programs in particular has been considered by Abdulla et al. [2] and Zhang et al. [17]. In our own previous work [10], we used Foata normal forms for generalized traces for generating representative executions of all four memory models of the SPARC hierarchy.

Chou and Peled [4] have formalized standard Mazurkiewicz traces in the context of formally verifying a partial-order reduction technique in HOL. Owens et al. [13], who formalized TSO in HOL4, pioneered formalization of semantic accounts of relaxed memory models with proof assistants.

## 6 Conclusion and Future Work

We believe it to be important to exercise care when choosing the semantic domain for behaviors for a class of concurrent systems. Descriptions of behaviors in terms of an apparently more involved abstraction can sometimes be more precise, yet still analyzable with less effort. In this paper, we presented certified Foata normalization of generalized Mazurkiewicz traces. The example from Sect. 4 demonstrates that standard Mazurkiewicz traces are not flexible enough and generalized traces can lead to fewer equivalence classes. That is good in any situation where one needs to exhaustively check an equivalence-invariant property on all equivalence classes.

In this paper, we looked at Foata normal forms. Another well-known normal form of traces is the lexicographic normal form. It would be interesting to see how this works with generalized traces. We also wonder whether something similar is possible for yet more flexible notions such as various specializations of pomsets. We are currently working on a generic framework for semantics and analysis for relaxed memory concurrency that has normal-forms based dynamic partial-order reduction built into its core and thus only deals with normal forms.

**Acknowledgements.** This work was supported by the ERDF funded Estonian national CoE project EXCITE and the Estonian Ministry of Education and Research institutional research grant IUT-3313.



## References

1. Aalbersberg, I.J.J., Rozenberg, G.: Theory of traces. *Theor. Comput. Sci.* **60**(1), 1–82 (1988)
2. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) *TACAS 2015*. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). [https://doi.org/10.1007/978-3-662-46681-0\\_28](https://doi.org/10.1007/978-3-662-46681-0_28)

3. Cartier, P., Foata, D.: *Problèmes combinatoires de commutation et réarrangements*. LNM, vol. 85. Springer, Heidelberg (1969). <https://doi.org/10.1007/BFb0079468>
4. Chou, C.-T., Peled, D.: Formal verification of a partial-order reduction technique for model checking. In: Margaria, T., Steffen, B. (eds.) *TACAS 1996*. LNCS, vol. 1055, pp. 241–257. Springer, Heidelberg (1996). [https://doi.org/10.1007/3-540-61042-1\\_48](https://doi.org/10.1007/3-540-61042-1_48)
5. Diekert, V., Métivier, T.: Partial commutation and traces. In: Rozenberg, G., Salomaa, A. (eds.) *Handbook of Formal Languages: Beyond Words*, vol. 3, pp. 457–553. Springer, Heidelberg (1997). [https://doi.org/10.1007/978-3-642-59126-6\\_8](https://doi.org/10.1007/978-3-642-59126-6_8)
6. Droste, M.: Concurrency, automata and domains. In: Paterson, M.S. (ed.) *ICALP 1990*. LNCS, vol. 443, pp. 195–208. Springer, Heidelberg (1990). <https://doi.org/10.1007/BFb0032032>
7. Godefroid, P.: Using partial orders to improve automatic verification methods. In: Clarke, E.M., Kurshan, R.P. (eds.) *CAV 1990*. LNCS, vol. 531, pp. 176–185. Springer, Heidelberg (1991). <https://doi.org/10.1007/BFb0023731>
8. Hoogers, P.W., Kleijn, H.C.M., Thiagarajan, P.S.: A trace semantics for Petri nets. *Inf. Comput.* **117**(1), 98–114 (1995)
9. Katz, S., Peled, D.: Defining conditional independence using collapses. *Theoret. Comput. Sci.* **101**(2), 337–359 (1995)
10. Maarand, H., Uustalu, T.: Generating representative executions. In: Vasconcelos, V.T., Haller, P. (eds.) *Proceedings of 10th Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software, PLACES 2017*. Electronic Processing Theoretical Computer Science, vol. 246, pp. 39–48. Open Publishing Association, Sydney (2017)
11. Mazurkiewicz, A.: *Concurrent program schemes and their interpretations*. DAIMI Report PB-78, Aarhus University (1977)
12. Norell, U.: Dependently typed programming in Agda. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) *AFP 2008*. LNCS, vol. 5832, pp. 230–266. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-04652-0\\_5](https://doi.org/10.1007/978-3-642-04652-0_5)
13. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) *TPHOLs 2009*. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-03359-9\\_27](https://doi.org/10.1007/978-3-642-03359-9_27)
14. Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) *CAV 1993*. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993). [https://doi.org/10.1007/3-540-56922-7\\_34](https://doi.org/10.1007/3-540-56922-7_34)
15. Sassone, V., Nielsen, M., Winskel, G.: Deterministic behavioural models for concurrency. In: Borzyszkowski, A.M., Sokołowski, S. (eds.) *MFCS 1993*. LNCS, vol. 711, pp. 682–692. Springer, Heidelberg (1993). [https://doi.org/10.1007/3-540-57182-5\\_59](https://doi.org/10.1007/3-540-57182-5_59)
16. SPARC International Inc.: *The SPARC Architecture Manual, Version 9*. Prentice Hall, Englewood Cliffs (1994). (Ed. by D.L. Weaver and T. Germond)
17. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: *Proceedings of 36th ACM SIGPLAN Conference on Principles of Language Design and Implementation, PLDI 2015*, pp. 250–259. ACM, New York (2015)



# On the Timed Analysis of Big-Data Applications

Francesco Marconi<sup>(✉)</sup> , Giovanni Quattrocchi, Luciano Baresi,  
Marcello M. Bersani, and Matteo Rossi 

DEIB, Politecnico di Milano, Milan, Italy  
{francesco.marconi,giovanni.quattrocchi,luciano.baresi,  
marcellomaria.bersani,matteo.rossi}@polimi.it

**Abstract.** Apache Spark is one of the best-known frameworks for executing big-data batch applications over a cluster of (virtual) machines. Defining the cluster (i.e., the number of machines and CPUs) to attain guarantees on the execution times (deadlines) of the application is indeed a trade-off between the cost of the infrastructure and the time needed to execute the application. Sizing the computational resources, in order to prevent cost overruns, can benefit from the use of formal models as a means to capture the execution time of applications.

Our model of Spark applications, based on the CLTL<sub>loc</sub> logic, is defined by considering the directed acyclic graph around which Spark programs are organized, the number of available CPUs, the number of tasks elaborated by the application, and the average execution times of tasks. If the outcome of the analysis is positive, then the execution is feasible—that is, it can be completed within a given time span. The analysis tool has been implemented on top of the Zot formal verification tool. A preliminary evaluation shows that our model is sufficiently accurate: the formal analysis identifies execution times that are close (the error is less than 10%) to those obtained by actually running the applications.

**Keywords:** Big-Data Applications · Metric temporal logic  
Formal verification · Apache Spark

## 1 Introduction

Many software systems produce huge quantities of data and their processing has been studied widely over the last years. Frameworks like Hadoop ([hadoop.apache.org](http://hadoop.apache.org)), Spark ([spark.apache.org](http://spark.apache.org)), and Flink ([flink.apache.org](http://flink.apache.org)), have been proposed to automate and ease the computation. These frameworks allow users to carry out batch processing over a cluster of (virtual) servers. The actual size of supplied data and the number of machines used impact the execution time by which the framework provides results. Unfortunately, the actual execution time is only known at the end of the computation, and estimations are mainly based on experience and domain-knowledge. In this context, guarantees over the quality of service are often stated as *deadlines*—i.e., the maximum

acceptable response times for single executions of the applications. The availability of tools that can foresee execution times, and thus help sizing the cluster, would greatly ease the adoption of these frameworks in contexts where time and costs are key drivers: the higher the cost (hence the more machines are available), the lower the overall response time.

The work presented in this paper is part of a larger research on a model-driven approach to the formal verification of Big Data frameworks carried out within the DICE project ([www.dice-h2020.eu](http://www.dice-h2020.eu)). In [15] we tackled the formal verification of data streaming applications based on the Storm framework. This paper focuses instead on Apache Spark, one of the best known frameworks for batch processing. Spark programs are internally represented as directed acyclic graphs (DAG) of operations. We propose the definition of formal models of Spark programs based on the CLTLoc [6] logic to allow for the validation of the required resources (virtual machines and CPU cores) given a deadline. A suitable formalization of the problem requires that the execution times of the different *tasks*—that is, of the different computation units—are properly modeled. Hence, we based the formal model on CLTLoc, a metric temporal logic over dense time that extends LTL with atomic constraints on clock variables. CLTLoc is supported by formal verification tools which allow users to analyze formulae in an automated manner [3, 6]. CLTLoc was also used—and extended—to model Storm topologies in [15]; this unified modeling and verification approach opens the possibility to analyze applications that are built upon heterogeneous building blocks, some tailored to stream processing, and others to batch processing.

The proposed solution builds the DAG-based representation of the program and automatically translates it into the corresponding CLTLoc model. The user then must provide the deadline, the number of available CPUs, the number of tasks elaborated by the application, and the average execution time of the different task types (e.g., obtained by profiling the program of interest). If the outcome of the analysis is positive, then the execution is feasible—that is, it can be completed by the given deadline. The prototype tool is implemented on top of Zot<sup>1</sup>, our verification tool for solving the bounded satisfiability problem for CLTLoc, and a first evaluation witnesses good prediction capabilities with an error that is usually less than 10%.

The rest of the paper is organized as follows: Sect. 2 introduces Spark and the CLTLoc logic; Sect. 3 presents the formal model; Sect. 4 discusses an experimental evaluation of the approach; Sect. 5 surveys related solutions, and Sect. 6 concludes.

## 2 Background

### 2.1 Apache Spark Framework

Spark is usually deployed on a cluster of servers and exploits a master/worker architecture. The *master* schedules operations for execution in the cluster by

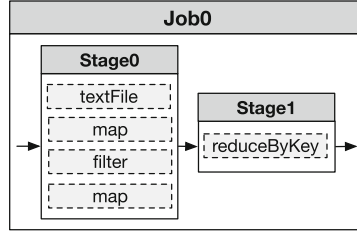
<sup>1</sup> [github.com/fm-polimi/zot](https://github.com/fm-polimi/zot).



```

spark.textFile("path/to/file")
.map(x => x.split(":"))
.filter(x => x(2) != "false")
.map(x => (x(0), x(1).toInt))
.reduceByKey(_ + _)
.collect()
    
```

(a) Code



(b) DAG

textFile	map	filter	map	reduceByKey
'a:2:true'	['a','2','true']	['a','2','true']	('a', 2)	
'e:3:true'	['e','3','true']	['e','3','true']	('e', 3)	('a', 7)
'a:3:false'	['a','3','false']	['u','1','true']	('u', 1)	('e', 3)
'u:1:true'	['u','1','true']	['u','5','true']	('u', 5)	('u', 6)
'o:4:false'	['o','4','false']	['a','5','true']	('a', 5)	
'u:5:true'	['u','5','true']			
'a:5:true'	['a','5','true']			

(c) Data

Fig. 1. Example of Spark application.

assigning part of the computation to each *worker*. The main programming abstraction in Spark is the *RDD* (resilient distributed dataset), i.e., immutable and fault-tolerant collections of homogeneous objects. An RDD is distributely stored into workers by means of multiple redundant partitions to facilitate parallel computation. The act of a worker to read from another worker’s memory or storage is called data shuffling. RDDs can be persisted in memory to improve performance through reuse. This makes Spark particularly efficient when executing iterative algorithms (e.g., machine learning and graph computations).

RDDs support two kinds of operations: *transformations* (e.g., map, filter) create new RDDs, while *actions* (e.g., count, collect) perform computations to generate values. The former are *lazy*: they are chained together for optimization purposes, and are performed only when an action is encountered. Spark distinguishes between *narrow* and *wide* transformations, where the former do not reshuffle data (e.g., map, filter), whereas the latter do (e.g., reduceByKey).

To fully comprehend how Spark works one must first understand how the logic of a particular application is broken down into parallelized tasks. Figure 1a shows the code (in Scala) of an example Spark application that performs a simple aggregation over a dataset read from a text file containing in each line a vowel, a number and a Boolean separated by colons. The goal of the program is to sum the numbers that are labeled with the same vowel which are also not marked as *false*. To do that the program chains different operations: (i) a *map* transforms each line in an array of strings by splitting it when a colon is encountered; (ii) a *filter* discards the unnecessary lines (those labeled with *false*); (iii) a second map converts the remaining arrays into key-value pairs, each one composed of a vowel (the key) and a number (the value); (iv) a *reduceByKey* is used to sum

the numbers that share the same key; finally (v) the dataset is returned using function *collect*. Figure 1c shows how an example dataset is transformed at each step. As soon as an application is submitted to Spark it is divided into multiple *jobs*. A job is a group of operations delimited by the presence of Spark actions within the code. When a job is scheduled for execution, a directed acyclic graph (DAG) of *stages* is created. Stages are delimited by operations that would require data shuffling, thus breaking data locality. Spark DAGs define the order among the stages of a job: two stages are connected if the second stage must read the data produced by the first, thus *a stage can be executed if and only if all of its predecessors are completed*. Once a stage is scheduled by the master, Spark defines the set of parallel tasks that need to be executed for the stage. A task executes all the transformations that compose a stage over a single partition of its input RDD. Tasks are executed in parallel by workers and are considered units of computation. Therefore, each task is executed by a single core and it is scheduled only when a core of a worker becomes free.

Figure 1b shows how logically the example program of Fig. 1a is executed by Spark. Each rectangle inside a stage is an RDD that is produced by performing the associated operation; the arrows define the ordering relation between stages Stage0 and Stage1 (i.e., a DAG made of two nodes executed in sequence). Due to the lazy evaluation of transformations nothing happens until *collect* is executed; at that moment Spark allocates a job by creating a DAG of stages. Because *map* and *filter* do not require data shuffling, the first four operations are grouped in a single stage (*Stage0*). Conversely, *reduceByKey* requires an exchange of data among workers since tuples with the same key are not guaranteed to be all in the same data partition. For this reason *Stage1* is created. *Stage1* depends on *Stage0* and so it can be scheduled only when the first has completed its execution.

## 2.2 Constraint LTL over-clocks

The temporal logic model of Sect. 3 is expressed in terms of the CLTLoc logic [6] augmented with discrete counters, an extension of LTL allowing clock variables and arithmetical variables to occur in atomic formulae.

Atomic formulae over  $(\mathbb{R}, \{<, =\})$  contain arithmetical variables, called *clock variables* (or simply clocks), which behave as clocks of Timed Automata [2]. A clock  $x$  measures the time elapsed since the last “reset” of  $x$ , which occurs when  $x = 0$  holds. Since the values of clocks can be compared with constants in formulae of the form  $x \sim c$  (where  $c \in \mathbb{N}$  and  $\sim \in \{<, =\}$ ), clocks are used to constrain the time elapsing between the events that characterize Spark computations.

Atomic formulae over  $(\mathbb{N}, \{<, =\}, +, 0, 1)$  predicate over arithmetical variables, called *counters*, that have no semantic restrictions. For instance, an atomic formula is  $y + z < 4$ , where both  $y$  and  $z$  are in  $\mathbb{N}$ . A counter stores a value that can be incremented, decremented and tested against a constant value. The logic exploits a special modality X applied to counters, that has been already introduced in [11], with the following meaning: if  $y$  is a counter,  $Xy$  is the value of  $y$  in the next position of time. Using modality X the increment of  $y$  by 1 is expressed

by the formula  $Xy = y + 1$  whereas  $y = Xy + 1$  indicates a decrement of  $y$  by 1. Counters are used in the model of Sect. 3 to represent the amount of tasks that are elaborated by Spark applications.

Let  $V$  be a finite set of variables over  $\mathbb{N}$ ,  $C$  a finite set of clock variables over  $\mathbb{R}$  and  $AP$  a finite set of atomic propositions. Atomic formulae  $\theta$  over  $V$  are quantifier-free Presburger formulae over terms  $\alpha$  of the form  $y$  or  $Xy$ , with  $y \in V$ . CLTLoc formulae  $\phi$  with counters are defined as:

$$\phi := p \mid x \sim c \mid \theta \mid \phi \wedge \phi \mid \neg\phi \mid \mathbf{X}\phi \mid \mathbf{Y}\phi \mid \phi\mathbf{U}\phi \mid \phi\mathbf{S}\phi$$

where  $p \in AP$ ,  $x \in C$ ,  $c \in \mathbb{N}$ ,  $\sim \in \{<, =\}$ , and  $\mathbf{X}$ ,  $\mathbf{Y}$ ,  $\mathbf{U}$  and  $\mathbf{S}$  are the usual “next”, “previous”, “until” and “since” operators of LTL. Operators  $\mathbf{F}$  (“eventually”),  $\mathbf{G}$  (“globally”), and  $\mathbf{P}$  (“previously”) are defined through the customary abbreviations:  $\mathbf{F}\phi = \top\mathbf{U}\phi$ ,  $\mathbf{G}\phi = \neg\mathbf{F}\neg\phi$ , and  $\mathbf{P}\phi = \top\mathbf{S}\phi$ .

An *interpretation* of a formula is a pair  $(\pi, \sigma)$ , where  $\pi : \mathbb{N} \rightarrow \wp(AP)$ , and  $\sigma : \mathbb{N} \times \{C \cup V\} \rightarrow \mathbb{R}$  is a mapping associating every variable in  $C \cup V$  with a value in  $\mathbb{R}$ , but restricting values of the elements in  $V$  to  $\mathbb{N}$ . The semantics of CLTLoc is defined as for LTL, except for formulae  $x \sim c$  and  $\theta$ . Let  $A_V$  be the ordered set of all terms of the form  $y$  and  $Xy$ , with  $y \in V$ , and let  $n$  be its cardinality; for each  $\alpha_j \in A_V$ , its depth  $|\alpha_j|$  is such that  $|\alpha_j| = 0$  if  $\alpha_j = y$ , and  $|\alpha_j| = 1$  if  $\alpha_j = Xy$  for some  $y \in V$ . Given a mapping  $v : A_V \rightarrow \mathbb{N}$ ,  $\theta[v(\alpha_0), \dots, v(\alpha_{n-1})]$  is the valuation of  $\theta$  through  $v$ , which is obtained by replacing each term  $\alpha_j$  occurring in  $\theta$  with value  $v(\alpha_j)$ . If  $\theta[v(\alpha_0), \dots, v(\alpha_{n-1})]$  holds, we write  $v \models \theta$ . Let  $t(\alpha_j) = y$  if  $\alpha_j$  is either  $y$  or  $Xy$ . The following properties hold for each  $i \in \mathbb{N}$ :

$$\begin{aligned} (\pi, \sigma), i \models x \sim c & \text{ iff } \sigma(i, x) \sim c \\ (\pi, \sigma), i \models \theta & \text{ iff } \theta[\sigma(i + |\alpha_0|, t(\alpha_0)), \dots, \sigma(i + |\alpha_{n-1}|, t(\alpha_{n-1}))] \end{aligned}$$

If  $\phi$  is a formula, interpretation  $(\pi, \sigma)$  is a *model* for  $\phi$  if  $(\pi, \sigma), 0 \models \phi$  holds.

The satisfiability problem CLTLoc is decidable [6] and can be practically computed through a Bounded Satisfiability Checking approach [3, 6]. Conversely, CLTLoc with Presburger arithmetics is undecidable, since so is its subset without clocks, CLTL [11], as the unboundedness of the domain of the counters and modality  $\mathbf{X}$  allow the logic to encode the computations of 2-counter machines. Even if our formal model of Spark computations is based on CLTLoc with counters, the value of arithmetical variables occurring therein is bounded by some value that depends on the problem instance (see Sect. 3). Therefore, the technique introduced in [3, 6] can still be exploited to solve the satisfiability problem for any instance of the model.

### 3 Modeling Spark Applications

This section presents the formal definition of the problem that we consider for the analysis of Spark applications and the temporal model that has been devised to solve it. Some assumptions are needed to abstract the Spark computation from details that are related to the physical infrastructure running the Spark framework and that depend on implementation aspects of the applications.

**Assumptions and Level of Abstraction.** We make the following assumptions.

*The cluster running the Spark application is composed of homogeneous machines.*

The workload of the cluster executing the application is not subject to oscillations that might alter the execution of the running jobs; hence, the performance of the cluster is stable and does not vary over time. The number of nodes in the cluster and the network latency are not explicitly represented in the model. However, they are strictly correlated as the more nodes are in the cluster, the higher the latency will be. For this reason, we decided to synthesize their effect as a single term to be included as an overhead to the task durations.

*Some features of the runtime environment of Spark are simplified;* for instance, the interaction among master and workers is not taken into account. The latency generated by the execution of services managing tasks is considered negligible with respect to the total execution time of the application.

*The input dataset provided to the application is homogeneous;* that is, the possible skewness of data is not taken into account. All tasks constituting a stage have durations that can vary non-deterministically by at most a fraction of a nominal value.

*The number of CPU cores that are available to the application is known* before starting the execution of the job and it does not vary over the computation.

*The functional aspects of executed operations are not directly considered in the model;* only their effect in terms of temporal behavior is represented.

The model is focused on the execution DAG underlying the application and it is based on an abstraction of the temporal behavior of stages and the tasks they are composed of. As explained in Sect. 2.1, the sequence of operations included in each stage is applied (possibly in parallel) on all partitions of the input dataset of the stage by means of a set of homogeneous tasks.

**Problem Statement.** Let  $D$  be a DAG  $(S, E)$  where  $S$  is a finite set of  $N$  stages  $\{S_0, \dots, S_{N-1}\}$  and  $E$  is a subset of  $S \times S$  representing the precedence relation among stages. Let  $\bar{T}_i$  be a finite set of homogeneous tasks associated with  $S_i$  such that any pair of tasks  $(\bar{T}_i, \bar{T}_{i'})$  are disjoint for any  $0 \leq i, i' < N$  (with  $i \neq i'$ ) and let  $\bar{T}$  be the set  $\bigcup_i \bar{T}_i$ . Hereafter, variables  $i, j$  are such that  $0 \leq i < N$  and  $0 \leq j < K$  hold.

An *execution*  $\eta$  of  $D$  with tasks in  $\bar{T}$  is a finite sequence of  $K$  tuples  $t^0, t^1, \dots, t^{K-1}$  of the form  $t^j = (T_0^j, \dots, T_{N-1}^j)$ , called *execution steps*, where each set of *active tasks*  $T_i^j$  is a—possibly empty—subset of  $\bar{T}_i$  satisfying the following constraints: (i) for every stage  $S_i$ , each task in  $\bar{T}_i$  appears in the execution sequence exactly once; also, if some task of  $\bar{T}_i$  occurs at step  $j$ , then all tasks associated with all stages  $S_{i'}$  preceding  $S_i$  with respect to  $E$  occur before  $j$ ; (ii) for each step there is at least one set of active tasks. A non-empty set  $T_i^j$  of tasks is called a *batch* of active tasks.

For any stage  $S_i$  in  $S$ , let  $\tau_i$  be a strictly positive constant in  $\mathbb{R}$  defining the time needed to compute a generic task of  $T_i$ . Let  $I$  and  $I'$  be two convex and bounded sets in  $\mathbb{R}$ . We say that  $I$  precedes  $I'$  when all the elements in  $I$  are strictly smaller than all the elements in  $I'$ . Given an execution  $\eta$  for  $D$ , define function  $active(t)$  specifying the set of active tasks of  $\bar{T}$  at any time instant  $t$ , such that for every  $t \in \mathbb{R}$ : (i) if a batch  $T_i^j$  is active at  $t$ , then there is an interval  $I$  of  $\tau_i$  time units, including  $t$ , where  $T_i^j$  is active and no task of  $T_i^j$  is active in any time instant  $t'$  not belonging to  $I$ ; (ii) every batch  $T_i^j$  is eventually active; (iii) if batch  $T_i^j$  occurs before batch  $T_i^{j'}$  in  $\eta$  (i.e.,  $j < j'$ ), then the interval of time where  $T_i^j$  is active precedes the interval of time where  $T_i^{j'}$  is active.

Given an integer  $p > 0$ , an execution  $\eta = t^0, t^1, \dots, t^{K-1}$  for  $D$  is *feasible* if  $|active(t)| \leq p$ , for all  $t \geq 0$ . The *time span*  $ts(\eta)$  of  $\eta$  is defined as the maximum time instant where at least one task in active.

The *feasibility problem* for a Spark application is defined as follows. Let  $D$  be a DAG  $(S, E)$  of  $N$  stages, let  $\bar{T}_i, \tau_i$  and  $p$  be defined as before and let  $d$  be a strictly positive integer. A solution of the feasibility problem for  $D$  with tasks in  $\bar{T}$  is a feasible execution  $\eta = t^0, t^1, \dots, t^{K-1}$  such that  $ts(\eta) < d$ . Let  $FD$  be the set of values  $\{d : \exists \eta \ ts(\eta) < d\}$  of the feasible deadlines, i.e., the set of all the possible deadlines  $d$  such that there exists a feasible execution whose duration is less than  $d$ . The *minimum feasible deadline (mfd)* is the minimum of  $FD$ .

Figure 2 shows a possible execution  $\eta$  for the DAG depicted in Fig. 2b whose stages  $S_1, S_2$  and  $S_3$  execute, respectively, 10, 21 and 15 tasks, grouped into the sets  $\bar{T}_1, \bar{T}_2$  and  $\bar{T}_3$ . Every rectangle represent a batch of running tasks and the number written therein is the size of the batch, i.e., the cardinality  $|T_i^j|$ . Stage 1 and 3 consists of two batches while Stage 2 is executed by means of 4 batches. The number of cores  $p$  is equal to 10, hence, in every time instant, the number of running tasks is limited by 10. Assuming that the time delay between  $T_1^1$  and  $T_1^2$  is 1.3 time units ( $\tau_2$  is 1 time unit), then the duration of the computation  $ts(\eta)$  is 26.3 time units.

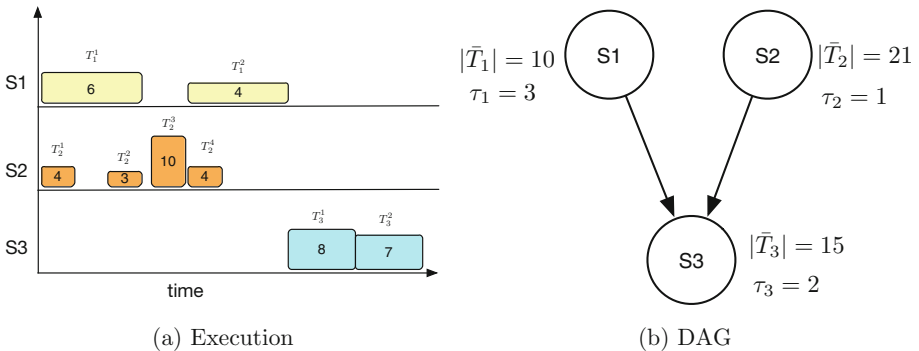
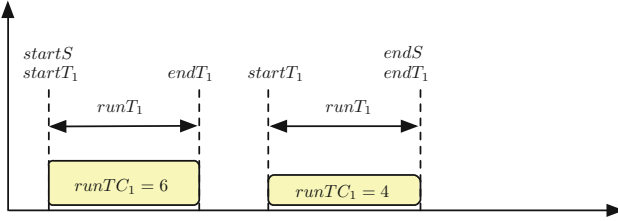


Fig. 2. Possible execution (2a) of the DAG in (2b).



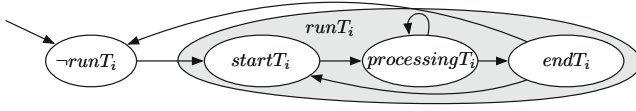
**Fig. 3.** Atomic propositions and discrete variables used to model the running batches and the overall computation of stage  $S_1$

**Temporal Logic Model of Spark Applications.** Consider a Spark execution DAG  $(S, E)$ . Suppose that the application is running on a cluster with  $p$  available cores, and each stage  $S_i$  is executed by running  $|\bar{T}_i|$  tasks. To represent the set of possible executions of the system, the CLTLoc model makes use of finite sets of atomic propositions, of discrete counters and of clocks. Atomic propositions are used to model the current status of stages and their tasks (i.e., whether they are started, running or completed), whereas the counters are used to keep track of the number of CPU cores that are either available, or are allocated to run the active tasks. Finally, the temporal constraints on the different tasks are expressed thanks to clocks.

Figure 3 shows the atomic propositions that are used to model the computation of the stage  $S_1$ , that is part of the DAG in Fig. 2b, according to the execution shown in Fig. 2a. Atoms  $\text{startS}_1$  and  $\text{endS}_1$  indicate the beginning and the end of the computation entailed by stage  $S_1$ , that is, the time instant where the first batch starts and the time instant where the last batch terminates. Batches of tasks are represented by means of  $\text{startT}_1$ ,  $\text{endT}_1$  and  $\text{runTC}_1$  that indicate, respectively, the beginning and the end of a batch and that the batch is currently active. The value of variable  $\text{runTC}_1$  is the number of tasks that are currently in execution, hence it corresponds to the value  $|\bar{T}_1^j|$ , for  $j \in \{1, 2\}$ , representing the cardinality of the active batch.

Corresponding to the three kinds of variables mentioned above, three groups of formulae can be identified in the model: those capturing the evolution of the state of stages and tasks; those constraining the number of tasks in execution with respect to the available cores; and the set of constraints on clocks. The three groups of formulae are presented in the rest of this section. Notice that all formulae presented in this section are implicitly universally quantified over time through the  $\mathbf{G}$  temporal operator.

*State formulae for stages.* A stage  $S_i$  can be either running (i.e., the atomic proposition  $\text{runS}_i$  holds) or not running. A stage becomes running—i.e.,  $\text{startS}_i$  holds—when there is at least one task that starts the execution and no task has been executed so far. If no tasks were executed then the number of tasks still to be processed, represented by discrete integer variable  $\text{remTC}_i$ , is equal to the total number of tasks that the stage has to elaborate ( $|\bar{T}_i|$ ). This situation is modeled through the following Formula (1).



**Fig. 4.** Finite state machine representing the state evolution of a set of tasks.

$$\bigwedge_{S_i \in S} (\text{startT}_i \wedge \text{remTC}_i = \|\bar{T}_i\| \iff \text{startS}_i) \quad (1)$$

A stage terminates—i.e.,  $\text{endS}_i$  holds—when there are no more tasks to be processed—i. e., when  $\text{remTC}_i$  is equal to 0. This is defined by Formula (2) below.

$$\bigwedge_{S_i \in S} (\text{endT}_i \wedge \text{remTC}_i = 0 \iff \text{endS}_i) \quad (2)$$

A stage is completed (i.e.,  $\text{completedS}_i$  holds) when it has been terminated in the past (i.e., there is a position before the current one where  $\text{endS}_i$  held); it is enabled (i.e.,  $\text{enabledS}_i$  holds) when all the predecessor stages  $S_j$ , such that  $(S_i, S_j)$  belongs to  $E$ , have been completed.

$$\bigwedge_{S_i \in S} (\text{completedS}_i \iff \mathbf{P}(\text{endS}_i)) \quad (3)$$

$$\bigwedge_{S_i \in S} (\text{enabledS}_i \iff \bigwedge_{S_j \in S, (S_i, S_j) \in E} \text{completedS}_j) \quad (4)$$

*State formulae for tasks.* The behaviour of each batch of tasks is summarized in Fig. 4. Initially, for each stage  $S_i$ , the corresponding batch of tasks is not running ( $\neg\text{runT}_i$  holds). In order for the batch to start processing ( $\text{runT}_i$  becomes true), the stage must be enabled (i. e.,  $\text{enabledS}_i$  holds), and some conditions on the resources (which are explained later, when describing counter-related formulae) must hold. Every execution of a batch is characterized by an initial state (in which  $\text{startT}_i$  holds) and a final state (in which  $\text{endT}_i$  holds).  $\text{processingT}_i$  is true in all time instants strictly included between the start and the end of a batch processing, and corresponds to  $\text{runT}_i \wedge \neg\text{startT}_i \wedge \neg\text{endT}_i$ . This execution cycle can be repeated many times depending on the available resources and the number of tasks to be executed. Being the batches of a stage sequential, they never overlap. Hence, atoms  $\text{runT}_i$ ,  $\text{startT}_i$  and  $\text{endT}_i$  are used to model any active batch  $T_i^j$ , as they can be safely reused to model all the batches required to complete a stage. For brevity, the CLTLoc formulae capturing the behavior of the state machine of Fig. 4 are not shown here.

*Counter-related Formulae.* Counter variables are used to define the constraints on system resources and the evolution of the tasks that are executed within the stage. For example, Formula (5) translates the constraint  $|\text{active}(t)| < p$ , for any

$t$ , given in the problem statement. It limits the number of cores that are allocated to execute the active tasks. In particular, the sum of the number of available (**avaCC**) and allocated cores is always equal to  $p$ , the number of cores that is assigned to the job. The number of the remaining tasks of a stage decreases during its execution: Formula (6) imposes that the next value of **remTC<sub>i</sub>** (i.e., **XremTC<sub>i</sub>**) is not greater than the value of **remTC<sub>i</sub>** in the current position.

$$\sum_{S_i \in S} (\text{runTC}_i) + \text{avaCC} = p \quad (5)$$

$$\bigwedge_{S_i \in S} (\text{remTC}_i \geq \text{XremTC}_i) \quad (6)$$

The following formulae link the truth value of the events **startT<sub>i</sub>** and **endT<sub>i</sub>** with the value of counters **runTC<sub>i</sub>** and **remTC<sub>i</sub>**. Formula (7) correlates variable **runTC<sub>i</sub>** with proposition **runT** by imposing that a batch is running (i.e., **runT** holds) when the value of **runTC<sub>i</sub>** of active tasks is strictly positive. The two formulae (8) and (9) determine the value of **runTC<sub>i</sub>** and **remTC<sub>i</sub>** during the execution of the batch. Since the model is not designed to represent core re-balancing operations, the formulae enforce a variation of **runTC<sub>i</sub>** or **remTC<sub>i</sub>** to occur when a batch starts or terminates. In particular, Formula (8) imposes that a variation of the value of **runTC<sub>i</sub>** between two adjacent positions is a sufficient condition to make **startT<sub>i</sub>** or **endT<sub>i</sub>** true. Therefore, between **startT<sub>i</sub>** and **endT<sub>i</sub>** **runTC<sub>i</sub>** cannot vary. Similarly, Formula (9) imposes that a variation of the value of **remTC<sub>i</sub>** is the sufficient condition to activate the execution of a batch (i.e., **startT<sub>i</sub>** holds). Finally, Formula (10) defines the relation between the variables **runTC<sub>i</sub>** and **remTC<sub>i</sub>**. It states that, if the execution of a batch of tasks is starting, the number **runTC<sub>i</sub>** of running tasks in the batch is the difference of the (number of) remaining tasks at the beginning of the batch (i.e., value **remTC<sub>i</sub>**) and the remaining tasks in the preceding position (i.e., value **YremTC<sub>i</sub>**).

$$\bigwedge_{S_i \in S} (\text{runT}_i \Leftrightarrow \text{runTC}_i > 0) \quad (7)$$

$$\bigwedge_{S_i \in S} ((\text{runTC}_i \neq \text{XrunTC}_i) \Rightarrow (\text{XstartT}_i \vee \text{endT}_i)) \quad (8)$$

$$\bigwedge_{S_i \in S} (\text{remTC}_i \neq \text{XremTC}_i \Rightarrow \text{XstartT}_i) \quad (9)$$

$$\bigwedge_{S_i \in S} (\text{startT}_i \Rightarrow (\text{runTC}_i = \text{YremTC}_i - \text{remTC}_i)) \quad (10)$$

*Constraints on clocks.* To represent the durations of events in the model, a clock variable **clock<sub>runT<sub>i</sub></sub>** has been defined for each stage  $S_i$ . Specifically, **clock<sub>runT<sub>i</sub></sub>** measures the duration of the **runT<sub>i</sub>** phases for each batch of tasks of stage  $S_i$ . The following formula defines the reset conditions for the clocks: **clock<sub>runT<sub>i</sub></sub>** is reset every time a new batch of tasks starts running for stage  $S_i$ .



$$\bigwedge_{S_i \in S} ((\text{clock}_{\text{runT}_i} = 0) \iff (\text{orig} \vee \text{startT}_i)) \quad (11)$$

Formula (12) limits the duration of the execution of a batch of tasks by imposing that the termination of the batch occurs when the value of clock  $\text{clock}_{\text{runT}_i}$  is in interval  $[\tau_i - \epsilon, \tau_i + \epsilon]$ , where  $\tau_i$  is the average task duration of stage  $S_i$  which is given as a parameter to the model, and  $\epsilon$  is a constant defining the variability in the processing duration with respect to  $\tau_i$ . If there is a batch currently running (i.e.,  $\text{runT}_i$  holds) then  $\text{runT}_i$  holds until an instant when the value of clock  $\text{clock}_{\text{runT}_i}$  is in  $[\tau_i - \epsilon, \tau_i + \epsilon]$  and  $\text{endT}_i$  is true.

$$\bigwedge_{S_i \in S} \left( \begin{array}{l} \text{runT}_i \Rightarrow \\ (\text{runT}_i \wedge \neg \text{endT}_i) \mathbf{U}((\text{clock}_{\text{runT}_i} \geq \tau_i - \epsilon) \wedge (\text{clock}_{\text{runT}_i} \leq \tau_i + \epsilon) \wedge \text{endT}_i) \end{array} \right) \quad (12)$$

*Initialization.* The initial condition of any modeled Spark application obeys the following constraints: (i) no tasks are running in the origin; (ii) for each stage  $S_i$ , the number of remaining tasks is  $|\bar{T}_i|$ ; (iii) the number of available cores  $\text{avaCC}$  is the total number of cores  $p$ .

## 4 Implementation and Validation of the Model

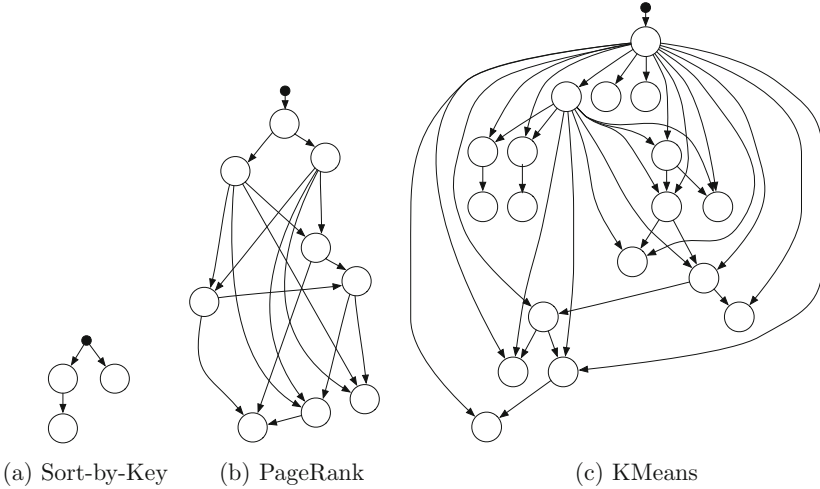
The goals of this section are twofold. First, it briefly introduces the prototype tool that automatically generates CLTLoc formal models from high-level descriptions of Spark DAGs. Second, it presents a set of experiments carried out with real-life Spark applications to evaluate the effectiveness of the approach. The validation focuses on understanding the accuracy with which the model is able to identify the actual deadline that can be met by an implemented application.

The implemented prototype tool, D-VerT<sup>2</sup>, takes as input a configuration file describing the Spark application to be analyzed, and uses a templating mechanism to automatically generate the corresponding formulae. The configuration file contains all the relevant information for running the analysis: the structure of the DAG, the number of tasks and the duration  $\tau_i$  for each stage  $i$ , the deadline against which the feasibility analysis has to be performed, the number of cores in the cluster and the number of time positions to be considered for running the verification. DAG structure and timing information can be either manually provided or automatically generated by means of a benchmarking tool<sup>3</sup> which, as explained later in this section, allows for the profiling of running applications and provides an estimation of the timing characteristics for different settings.

D-VerT produces the corresponding instance of the formal model of Sect. 3 in the input format of the Zot verification tool, which is able to analyze CLTLoc formulae, and feeds the model to Zot. It then collects the outcome of the formal analysis and provides, when possible, a graphical representation of the results

<sup>2</sup> [github.com/dice-project/DICE-Verification](https://github.com/dice-project/DICE-Verification).

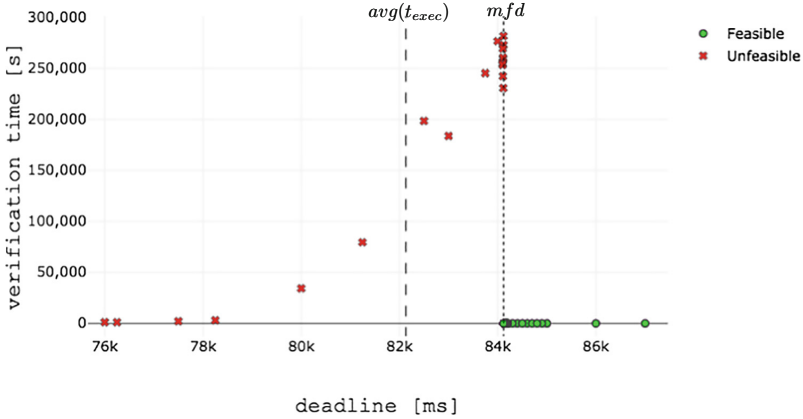
<sup>3</sup> [github.com/franco-maroni/xSpark-bench](https://github.com/franco-maroni/xSpark-bench).



**Fig. 5.** DAGs of selected applications.

for better readability. The use of a declarative, logic-based modeling approach facilitates this automatic process, since the formulae are easily generalizable to any kind of DAG structure. Further details on the D-VerT toolchain can be found in [5].

We selected three well-known applications to perform the analysis and evaluate it against realistic use cases: the simple *SortByKey* operation; the graph processing algorithm *PageRank* [8]; and the clustering procedure *K-Means* [14]. As depicted in Fig. 5, the execution DAG of the three use cases have different size and level of complexity. To evaluate the model with respect to a variety of scenarios, for each one of these applications we selected six different settings in terms of both the configuration of the underlying cluster (i.e., two different numbers of available cores for each cluster node), and the configuration of the single application (i.e., three different dimensions of the input dataset and same number of partitions used for each stage). Next, we performed a profiling activity that consisted in launching several times the different applications using two different versions of Spark: one, called from now on *sequential Spark*, was slightly modified by us and the other was the regular version of Spark (i.e., *vanilla Spark*). For both cases we collected the timing information of all the stages and tasks. Our modifications in *sequential Spark* force the scheduler to launch all the stages sequentially (i.e., no more than one stage can be simultaneously in execution), allowing us to cleanly isolate the durations of each stage and its tasks, without the noise introduced by the concurrent execution of multiple stages. These durations were used to automatically generate the configuration files, therefore to instantiate the formal model in its different settings. On the other hand, the *average execution times* of the entire applications collected on *vanilla Spark* (from now on  $avg(t_{exec})$ ), were used as the reference against which



**Fig. 6.** Times and outcomes of the verification tasks on the *SortByKey* use case (22 cores, 100 tasks and 300M input records) by providing different deadlines.

to compare the results of the analysis. We performed various verification tasks on each instance of the formal model to identify, for each configuration, the estimated set of feasible deadlines *FD*. Once the set was identified, we compared the *minimum feasible deadline (mfd)* found with the corresponding  $avg(t_{exec})$ , and we used the difference between them to evaluate the accuracy of the model (expressed as the percentage error *err*).

The first use case we considered is *SortByKey*. After an extensive analysis by means of multiple verification runs (each of them with a different deadline), we were able to identify the feasibility sets and the minimum feasible deadline, considering the granularity of the milliseconds. Figure 6 shows a comprehensive view of the verification tasks, performed on a single setting of the use case, with their outcomes (feasible/unfeasible) and the corresponding verification times. The *mfd* found was 84120 ms, therefore all the deadlines higher than that are feasible. On the other hand, deadlines of 84119 and below resulted unfeasible. Since, for this setting,  $avg(t_{exec})$  was 82133 ms, the percentage error *err* is about 2.4%. This analysis highlighted a strong dependency of the verification time on the closeness of the analyzed deadline to the minimum feasible deadline. As reported in Fig. 6, verification time is in the order of the seconds for all deadlines lower than 75000 ms or greater than or equal to 84120 ms (*mfd*), whereas it grows exponentially for increasing deadline values between 75000 ms and 84119 ms, peaking at around 78 h for 84117 ms. The notable growth is therefore registered for those deadlines that resulted unfeasible, but close to *mfd*. This pattern has been observed also for the other, more complex, applications we analyzed. However, since the verification times grow significantly with the size of the DAG (the analysis for feasible deadlines is generally completed in the order of minutes for *PageRank* and in the order of hours for *K-Means*) the time needed to perform the verification of some unfeasible deadlines becomes unmanageable in practice. Therefore, since there is such a pronounced difference between the times for fea-

**Table 1.** Experimental results (full experimental data available at [10.5281/zenodo.1162853](https://zenodo.org/record/1162853)).

<i>app</i>	<i>cores</i>	<i>tasks</i>	<i>records<sub>in</sub></i>	<i>avg(t<sub>exec</sub>)</i> (ms)	<i>mfd</i> (ms)	<i>err</i>
<i>SortByKey</i>	12	100	260M	88386	91384	3.3%
			280M	100769	98420	2%
			300M	107054	105443	1.5%
	22	100	260M	74919	72904	2.6%
			280M	77884	78500	0.7%
			300M	82133	84120	2.4%
<i>PageRank</i>	28	128	200M	60028	62500	4%
			300M	87787	94000	7%
			400M	116810	120000	2%
	48	128	200M	48805	47000	3.6%
			300M	66636	65100	2.3%
			400M	88320	86000	2.6%
<i>K-Means</i>	24	18	80M	77651	79000	1.7%
			120M	103492	107000	3%
			160M	131600	140000	6%
	32	24	80M	64565	63000	2%
			120M	81299	82000	1%
			160M	101483	103000	1%

sible results and the times for unfeasible deadlines in the neighborhood of *mfd*, we pursued the following heuristic approach: for each configuration we started by running the analysis for trivially feasible deadlines and then proceeded “backwards” (i. e., by lowering the deadline) until a strong discontinuity was found in the verification time. Based on the times registered for each feasible deadline, we defined some timeouts and concluded that a given deadline was reasonably not feasible if no result was returned by the tool within those timeouts. Table 1 shows the experimental findings of the validation activity for the three applications. Each row represents a different application setting, characterized by a specific number of cores in the cluster, a number of tasks (i. e., partitions) for each stage, and a dimension of the input dataset in terms of number of records (*records<sub>in</sub>*). The measures of interests are the previously defined *avg(t<sub>exec</sub>)*, *mfd* and the related percentage error *err*.

Results show that adherence of the model to the actual execution times with *vanilla* Spark (i. e., of *mfd* to *avg(t<sub>exec</sub>)*) is not particularly affected by changes in the use case type and configuration. In fact, *err* is at most 4% across all 6 settings of *SortByKey*, at most 7% for *PageRank* and at most 6% for *K-Means*.

## 5 Related Works

To the best of our knowledge, no approaches exist in literature for the formal verification of Spark applications. For this reason, we cannot directly compare against other works having the same focus. In the following, we present other techniques, in some cases applied to distributed systems, that tackle problems somewhat similar to ours, starting with general scheduling problems.

The analysis of temporal properties of scheduling algorithms and of distributed systems has been addressed with positive outcomes by using Timed Automata (TA, [2]) and Hybrid Automata (HA, [12]). In [10], TA are used for the analysis of the task scheduling of Ada programs, in systems equipped with one CPU that executes both the scheduler and the Ada code. Unlike in standard schedulability analysis (e.g., [16]), the use of TA—and, similarly, the use of CLTLoc in the present work—allows for capturing relevant properties of real implementations (e.g., resource constraints), and for the relaxing of some restrictions on the software structure, that are needed for the analysis. A timed analysis for distributed systems has been addressed in [7] by means of HA. HA model the execution of concurrent tasks on the available CPUs and the precedence relation among the tasks, which is specified by a graph of dependencies. The tasks are indivisible units of work with a fixed duration, they have a scheduling priority and can be preempted. [13] also uses TA to model distributed real-time applications. A distributed application in [13] consists of several concurrent tasks, each one running on a single processor and communicating with the others via a network. TA are used to model the interaction among the tasks, the network (sender and receiver component) and the arbiter of the communication channel. Both the schedulability of the tasks and the application response-time are analyzed by using a state-of-the-art model-checker for TA and for HA. Our model considers DAG of stages similar to the graph of dependencies in [7]. However, whereas tasks in [7, 13] are atomic and are executed on a single CPU each, the execution of a Spark stage can be spread over different CPUs, complicating the model.

Operations Research (OR) offers a wide range of techniques for scheduling and planning problems. TA and their extensions are very effective tools to tackle non-standard problems that cannot be solved by using standard OR techniques. [4] presents Priced TA (PTA), which extend TA with costs and are suitable for modeling scheduling problems with optimal goals. PTA allow for computing the minimum optimal cost of reaching a target configuration. Three standard problems of OR are dealt with PTA and the experimental results, comparing the standard MILP-based approaches with the PTA algorithm, indicate that PTA are competitive and, in some cases, faster. The Job-shop problem, that [4] addresses by means of PTA, and the extension with bounded delay uncertainty are addressed in [1] by using standard TA. The experimental results again demonstrate that the TA-based procedures applied to the problem can provide better outcomes, that is, more efficient schedules, than those produced with standard OR algorithms.

As shown in [6], CLTLoc has the same expressive power as TA. Hence, in principle any problem solved through TA can also be solved through CLTLoc, and vice-versa. The CLTLoc-based approach that we pursue in this work allows for a high degree of modularity in the generation of the formal model from its high-level description, as it is easy to focus on the various aspects of the model (e.g., precedences among stages, timing and resource constraints) one at a time—each aspect corresponding to a different set of logic constraints. In addition, as mentioned in Sect. 1, CLTLoc is the basis for a unifying approach to the modeling of Big Data frameworks which tackles applications of different natures (stream vs. batch processing).

In the domain of the analysis of Big Data frameworks, simulation, rather than formal verification is usually the approach of choice. For example, [17] considers the problem of computing the response-time of a Spark application through simulation of a Stochastic Petri Net (SPN) model. The experimental results demonstrate that the error affecting the simulation is low (less than 10%) when the simulated application has a high number of tasks and cores (e.g., more than 12 cores and 200 tasks). For some configurations, however, an error bigger than 30% is possible. In [9] an ad-hoc fast event driven simulator, called *dagSIM*, has been used to simulate applications modeled as DAGs of nodes representing the execution of batches of tasks whose average duration is described with a stochastic distribution. DagSIM predicts the application response time by means of a resolution procedure which is faster than the one based on SPN. However, simulation-based approaches—unlike verification-based ones—cannot offer *guarantees* about the feasibility or not of a desired deadline, and in particular they cannot be used to determine the *unfeasibility* of a deadline.

As already mentioned in the introduction, an analogous—temporal logic-based—approach was followed in [15] for the analysis of Storm applications. This work and [15] are based on the same automated mechanism implemented in the D-VerT tool. However, the formal model presented in [15] represents a different computation paradigm, namely, the stream processing, by means of CLTLoc extended with discrete *unbounded* counters. The analyses performed in the two works are different as well: [15] aims at finding ultimately periodic traces witnessing the presence of bottlenecks in the application, while this work focuses on finding finite traces proving the feasibility of given deadlines.

## 6 Conclusion

This work proposed an approach and a prototype tool to formally verify the feasibility of satisfying constraints over the response time of Spark applications given a fixed amount of computational resources. An experimental evaluation shows promising results in terms of accuracy of the model with respect to real Spark executions on different use cases and settings.

Possible future works include: (i) undertaking a thorough analysis of the complexity of the model and its effects on the verification times; (ii) improvements of the verification performance by optimizing the formal model; (iii) a refinement of the profiling phase aimed at providing good estimates of the execution times against changes in the number of cores and partitions of the input dataset.

**Acknowledgment.** This work has been partially supported by the DICE project (Horizon 2020 project no. 644869) and by the GAUSS national research project (MIUR, PRIN 2015, Contract 2015KWREMX).

## References

1. Abdeddaim, Y., Asarin, E., Maler, O.: Scheduling with timed automata. *Theoret. Comput. Sci.* **354**(2), 272–300 (2006)
2. Alur, R., Dill, D.L.: A theory of timed automata. *Theoret. Comput. Sci.* **126**(2), 183–235 (1994)
3. Baresi, L., Pourhashem Kallehbasti, M.M., Rossi, M.: How bit-vector logic can help improve the verification of LTL specifications over infinite domains. In: *Proceedings of 31st Annual ACM Symposium on Applied Computing*, pp. 1666–1673 (2016)
4. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Optimal scheduling using priced timed automata. *SIGMETRICS Perform. Eval. Rev.* **32**(4), 34–40 (2005)
5. Bersani, M., Erascu, M., Marconi, F., Rossi, M.: DICE verification tool - final version. Technical report, DICE Consortium (2017). [www.dice-h2020.eu](http://www.dice-h2020.eu)
6. Bersani, M.M., Rossi, M., San Pietro, P.: A tool for deciding the satisfiability of continuous-time metric temporal logic. *Acta Informatica* **53**(2), 171–206 (2016)
7. Bradley, S., Henderson, W., Kendall, D.: Using timed automata for response time analysis of distributed real-time systems. In: *24th IFAC/IFIP Workshop on Real-Time Programming*, pp. 143–148 (1999)
8. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: *Proceedings of International World-Wide Web Conference (WWW)*, pp. 107–117 (1998)
9. Brito, A., Ardagna, D., Blanquer, I., Evangelinou, A., Barbierato, E., Gribaudo, M., Almeida, J., Couto, A.P., Braga, T.: D3.4 EUBra-BIGSEA QoS infrastructure services intermediate version. Technical report, 3 July 2017
10. Corbett, J.C.: Timing analysis of ada tasking programs. *IEEE Trans. Softw. Eng.* **22**(7), 461–483 (1996)
11. Demri, S., D’Souza, D.: An automata-theoretic approach to constraint LTL. *Inf. Comput.* **205**(3), 380–415 (2007)
12. Henzinger, T.A.: The theory of hybrid automata. In: Inan, M.K., Kurshan, R.P. (eds.) *Verification of Digital and Hybrid Systems*. NATO ASI Series (Series F: Computer and Systems Sciences), vol. 170, pp. 265–292. Springer, Berlin (2000). [https://doi.org/10.1007/978-3-642-59615-5\\_13](https://doi.org/10.1007/978-3-642-59615-5_13)
13. Krakora, J., Waszniowski, L., Pisa, P., Hanzalek, Z.: Timed automata approach to real time distributed system verification. In: *Proceedings of IEEE International Workshop on Factory Communication Systems*, pp. 407–410, September 2004
14. MacQueen, J., et al.: Some methods for classification and analysis of multivariate observations. In: *Proceedings of Berkeley symposium on mathematical statistics and probability*, vol. 1, pp. 281–297 (1967)

15. Marconi, F., Bersani, M.M., Erascu, M., Rossi, M.: Towards the formal verification of data-intensive applications through metric temporal logic. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM 2016. LNCS, vol. 10009, pp. 193–209. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-47846-3\\_13](https://doi.org/10.1007/978-3-319-47846-3_13)
16. Palencia, J.C., Harbour, M.G.: Schedulability analysis for tasks with static and dynamic offsets. In: Proceedings of IEEE Real-Time Systems Symposium, pp. 26–37, December 1998
17. Perez, D., Bernardi, S., Merseguer, J.Z., Requeno, J.I., Casale, G., Zhu, L.: DICE simulation tools - final version. Deliverable. <http://www.dice-h2020.eu/resources/>





# Tuning Permissiveness of Active Safety Monitors for Autonomous Systems

Lola Masson<sup>1</sup>(✉), Jérémie Guiochet<sup>1,2</sup>, H el ene Waeselynck<sup>1</sup>, Kalou Cabrera<sup>1</sup>,  
Sofia Cassel<sup>3</sup>, and Martin T orngr en<sup>3</sup>

<sup>1</sup> LAAS-CNRS, CNRS, Toulouse, France

{lola.masson,jeremie.guiochet,helene.waeselynck,kalou.cabrera}@laas.fr

<sup>2</sup> Universit e de Toulouse, UPS, Toulouse, France

<sup>3</sup> KTH, Stockholm, Sweden

{sofia.cassel,martin}@md.kth.se

**Abstract.** Robots and autonomous systems have become a part of our everyday life, therefore guaranteeing their safety is crucial. Among the possible ways to do so, monitoring is widely used, but few methods exist to systematically generate safety rules to implement such monitors. Particularly, building safety monitors that do not constrain excessively the system's ability to perform its tasks is necessary as those systems operate with few human interventions. We propose in this paper a method to take into account the system's desired tasks in the specification of strategies for monitors and apply it to a case study. We show that we allow more strategies to be found and we facilitate the reasoning about the trade-off between safety and availability.

## 1 Introduction

Autonomous systems are becoming an increasing part of our daily lives: medical robots, self-driving cars, and industrial robots are good examples. It is critical to be able to guarantee the safety of such systems, since they operate independently in the vicinity of humans. One way to guarantee safety of autonomous systems is to attempt to specify them completely and reason about any dangerous behavior before deployment. This, however, requires that they behave predictably in any situation, which is not true as the systems interact with other humans or autonomous systems, in unstructured environments. In these cases, *monitoring* is an attractive option for guaranteeing safety. It consists in delegating the safety aspect to an independent component. It facilitates the task since a detailed specification of all possible behaviors is not necessary. A coarse granularity is sufficient to distinguish between safe and unsafe behavior. A *safety monitor* watches the system in operation, and intervenes as soon as potentially dangerous behavior is detected, typically by inhibiting certain actions, or by triggering corrective actions.

Such an approach, that only studies unsafe behaviors, may however restrict the system to the point where it cannot function in a meaningful way, i.e. availability suffers. For example, a robot having to transport objects might be kept

standing still, or prohibited from dropping or picking up anything. The system might be safe, but useless for its purposes.

In this paper, we address the problem of monitoring autonomous systems while avoiding too conservative restrictions on behavior. This refers to the classical trade-off between safety and availability (readiness for usage). The system's availability depends on what we call the monitor's *permissiveness*, i.e. its ability to let the system reach functional states. We specify permissiveness by identifying the behaviors that are essential to the system's function. The specification is introduced as an extension to SMOF [15], a Safety Monitoring Framework we developed for the synthesis of safety strategies. It aims to facilitate the tuning of the monitor's permissiveness and the reasoning about how safety and availability interrelate in the context of a particular system.

The paper is organised as follows: in Sect. 2, we give an overview of related work; we detail the background of this work in Sect. 3; in Sect. 4, we present how we express functionality requirements for the synthesis of safety strategies. We apply this method to an example in Sect. 5. We conclude in Sect. 6.

## 2 Related Work

Safety monitoring is a common mechanism used as part of fault-tolerance approaches [4] usually implemented as an independent mechanism that forces the system to stay in a safe state. This principle has been used in robotics under different names such as *safety manager* [18], *autonomous safety system* [19], *guardian agent* [7], or *emergency layer* [9].

In all these works, the specification of the safety strategies (i.e. the rules that the monitor follows to trigger safety interventions) is essentially done ad hoc: the user would manually choose what is the best (combination of) action(s) to trigger to avoid a risk, and when to do so. Other authors provide methods to identify safety invariants either from a hazard analysis [21] or from execution traces [11]. Some specify safety strategies in a DSL (Domain Specific Language) in order to generate code [3, 10]. But none of them offers a complete approach to identify invariants from hazards and formally derive the safety strategies. In contrast, our previous work [14, 15] provides a complete safety rule identification process, starting from a hazard analysis using the HAZOP-UML [8] technique and using formal verification techniques to synthesize the strategies.

Safety monitoring is related to runtime verification and property enforcement. Runtime verification [5, 12] checks for properties (e.g., in temporal logic) by typically adding code into the controller software. Property enforcement [6] extends runtime verification with the ability to modify the execution of the controller, in order to ensure the property. These techniques consider a richer set of property classes than safety ones, and, most importantly, can be tightly coupled to the system. It makes the underlying mechanisms quite different from the external safety monitors considered in this paper which have to rely on limited observation and intervention means. Though, the *transparency* property mentioned for example in [13, 16] for the specification of runtime monitors is close

to our permissiveness property: the runtime monitor should not modify already correct behaviors.

### 3 Baseline and Concepts

Our method to synthesize safety strategies for monitors is presented in [15]: SMOF (Safety Monitoring Framework). We briefly explain the SMOF principles below and introduce the motivation for the extension proposed in this paper.

#### 3.1 Safety Invariants, Margins and States

As a first step of the process, one identifies a list of hazards that may occur during the system’s operation, using the model-based hazard analysis HAZOP-UML [8]. From the list of hazards, one extracts those that can be treated by the monitor. Those hazards are reformulated as *safety invariants* such that each hazard is represented by the violation of an invariant. A safety invariant is a logic formula over a set of observable variables, derived from sensor values. The formalization of hazards as invariants may reveal the need for additional observation means, like in [17] where the original system design was revised to add sensors.

A combination of observation values defines a system state, as perceived by the monitor. If one of the safety invariants is violated, the system enters a *catastrophic state* that is assumed irreversible. Each safety invariant partitions the state space into catastrophic and non-catastrophic states as represented in Fig. 1. The non-catastrophic states can in turn be partitioned into *safe* and *warning states*, in such a way that any path from a safe state to a catastrophic one traverses a warning state. The warning states correspond to safety margins on the values of observations.

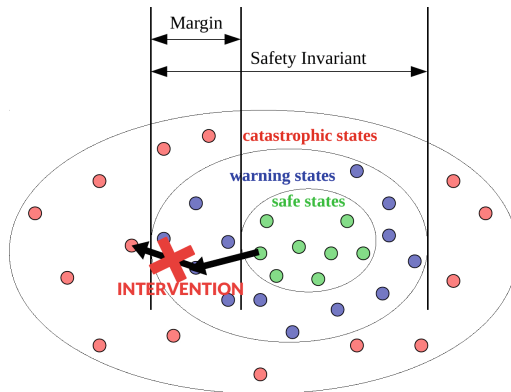


Fig. 1. System state space from the perspective of the monitor (Color figure online)

The monitor has means to prevent the evolution of the system towards the catastrophic states: these means are a set of safety *interventions* (mostly based

on actuators) made available to it. An intervention is modeled by its effect (constraints that cut some transitions) and preconditions (constraint on the state in which it can be applied). Interventions are applied in warning states in order to cut all the existing transitions to the catastrophic states, as shown in Fig. 1 by the red cross. The association of interventions to warning states constitutes a *safety strategy*. For example, let us assume that the invariant involves a predicate  $v < V_{max}$  (the velocity should always be lower than  $V_{max}$ ). In order to prevent evolution towards  $V_{max}$ , the strategy will associate one or several intervention(s) to warning states corresponding to a velocity higher than the threshold  $V_{max} - margin$ . The determination of the size of the margin involves a worst-case analysis, accounting for the dynamics of the physical system, as well as for the detection and reaction time of the monitor after the threshold crossing.

### 3.2 Safety and Permissiveness Properties

The safety strategy must fulfill two types of properties: *safety* and *permissiveness* properties. Both properties are expressed using CTL (Computation Tree Logic) which is well suited for reachability properties. *Safety* is defined as the non reachability of the catastrophic states. *Permissiveness* properties are intended to ensure that the strategy still permits functionality of the system, or, in other words maintains its availability. This is necessary to avoid safe strategies that would constrain the system's behavior to the point where it becomes useless (e.g., always engaging brakes to forbid any movement). SMOF adopts the view that the monitored system will be able to achieve its tasks if it can freely reach a wide range of states (e.g., it can reach states with a non zero velocity). Accordingly, permissiveness is generically formulated in terms of state reachability requirements: every non-catastrophic state must remain reachable from every other non-catastrophic state. We call it *universal permissiveness*. The safety strategy may cut some of the paths between pairs of states, but not all of the paths. In CTL, this is expressed as:  $AG(EF(nc\_state))$ , for each non-catastrophic state. Indeed, EF specifies that the state of interest is reachable from the initial state, and AG extends this to the reachability from every state. The user can also use the *simple permissiveness* which merely requires the reachability from the initial state:  $EF(nc\_state)$ . It is much weaker than the universal permissiveness as it allows some of the monitor's interventions to be irreversible: after reaching a warning state in which the monitor intervenes, the system may be confined into a subset of states for the rest of the execution. For example, an emergency stop can permanently affect the ability of the system to reach states with a non zero velocity.

### 3.3 SMOF Tooling

The SMOF tool support [2] includes the synthesis algorithm and a modelling template to ease the formalization of the different elements of the model: the behavior model with a partition into safe, warning and catastrophic states; the available interventions modeled by their effect on observable state variables; the

safety and permissiveness properties. The template offers predefined modules, as well as auto-completion facilities. For example, the tool automatically identifies the set of warning states (having a transition to a catastrophic state). Also, the permissiveness properties are automatically generated based on the identification of non-catastrophic states. Finally, SMOF provides a synthesis tool based on the model-checker NuSMV [1]. For this reason the NuSMV language is used for the formalization and we will use the `typewriter` font to refer to it in the rest of the paper. The SMOF synthesis tool relies on a branch and bound algorithm that associates interventions to warning states and checks some criteria to evaluate if the branch should be cut or explored. It returns a set of both safe and permissive strategies for the given invariant to enforce (see [15] for details).

The formalization and strategy synthesis is done for each invariant separately. Then a last step is to merge the models and to check for the consistency of the strategies selected for the different invariants.

The SMOF method and tool have been applied to real examples of robots: an industrial co-worker in a manufacturing setting [15], and a maintenance robot in airfield [17]. Examples and tutorials can be found online [2].

### 3.4 On Tuning Permissiveness Properties

This paper revisits the notion of permissiveness, in order to address some limitations of the generic definition adopted in SMOF. By default, SMOF requires the universal permissiveness, which is a very stringent requirement. As a result, the synthesis algorithm prunes any strategy that would cut all paths to a non-catastrophic state, even though this specific state may be useless for the accomplishment of the functions of the system. To give an example, let us consider a classical invariant stating that the system velocity should never reach a maximal absolute value  $V_{max}$ . The synthesis would reject any strategy preventing reachability of warning states with values close to  $V_{max}$ . But the cruise velocity of the system, used to accomplish its functions, is typically much lower than  $V_{max}$  and  $V_{max} - margin$ . Requiring the universal reachability of the warning states is useless in this case, since getting close to  $V_{max}$  is not a nominal behavior. The system operation could well accommodate a safety strategy that forbids evolution to close-to-catastrophic velocity values.

Suppose now that we do not find any solution respecting universal permissiveness. The user can choose to require the simple permissiveness. The requirements would be dramatically weakened. We would accept strategies that permanently affect the reachability of *any* non-catastrophic state, e.g., not only the close-to-catastrophic velocity values but also the moderate ones.

From what precedes, it may seem that we could simply modify the generic definition of permissiveness to require universal reachability of safe states only, excluding warning states. However, this would not work for all systems, as demonstrated by the maintenance robot studied in [17]. For this robot, some warning states *do* correspond to a nominal behavior and are essential to the accomplishment of the maintenance mission. More precisely, the robot is intended to control the intensity of lights along the airport runways. The light

measurement task is done by moving very close to the lights, which, from the perspective of the anticollision invariant, corresponds to a warning state. Any safety strategy removing reachability of a close-to-catastrophic distance to the lights would defeat the very purpose of the robot.

Actually, there is no *generic* definition of permissiveness that would provide the best trade-off with respect to the system functions. We would need to incorporate some *application-specific* information to tune the permissiveness requirements to the needs of the system. This paper proposes a way to introduce such custom permissiveness properties into SMOF, allowing more strategies to be found and facilitating the elicitation of trade-offs in cases where some functionalities must be restricted due to safety concerns.

## 4 Defining Custom Permissiveness Properties

The custom permissiveness properties are introduced by a dedicated state model, focusing on the identification of the states that are essential to the system functionalities (Sect. 4.1). The essential/not essential view is different from the safe/warning/catastrophic one we have in the safety state model. We thus need to bind together the functionality and safety models to reflect the fact that they represent two orthogonal decompositions of the same system state space (Sect. 4.2). Once this is done, custom permissiveness properties can be used as a replacement for the generic ones: the synthesis tool will search for safe strategies ensuring universal permissiveness with respect a subset of non-catastrophic states, the ones identified as essential. In case ignoring the non-essential states does not suffice to allow a strategy to be found, the user may consider whether restricting one of the functionalities is feasible (Sect. 4.3). The whole approach does not essentially change the principles of SMOF, which should facilitate its integration into the existing toolchain (Sect. 4.4).

### 4.1 A Formal Model for the Permissiveness

We consider that a functionality is defined by a goal or objective that the system was designed to achieve. For example, if the system is designed to pick up objects and transport them, two of its functionalities could be “move from A to B” and “pick up an object”. Some of the functionalities are not related to any of the monitored variables, and therefore do not need to be considered.

To be used in the synthesis, we must model the permissiveness properties associated to the identified functionalities. While generic permissiveness properties apply to all non-catastrophic states, we choose to express the custom ones as the reachability of a subset of states, the ones that are essential to the system’s functionalities.

The state model for the functionalities is defined as a set of variables partitioned in classes of values of interest. For instance, let us consider the functionality  $f$ , which requires observable variable  $v$  (e.g., velocity, or position of a tool)

to reach a given value  $V_{\text{req}}$  (e.g., cruise velocity, vertical position) with some tolerance  $\delta$ . The domain of the variable  $v$  would be partitioned into three classes: 0 corresponding to values lower than  $V_{\text{req}} - \delta$ ; 1 to values in  $[V_{\text{req}} - \delta, V_{\text{req}} + \delta]$ ; 2 to values greater than  $V_{\text{req}} + \delta$ .

Let  $v_f : \{0, 1, 2\}$  be the abstract variable encoding the partition from the point of view of the functionality. The SMOF template provides a predefined module to express the continuity constraints on the evolution of the variables values. For example, the velocity cannot jump from 0 to 2 without traversing the nominal range of values. Generally speaking, the modeling can reuse the syntactic facilities offered for the safety model, also defined in terms of observable variables, classes of values of interest and evolution constraints.

The purpose of the functionality model is to allow the user to conveniently specify sets of states that are essential to a given functionality. A set of states is introduced by a predicate `req` over a variable or a set of variables. In the above example, the user would specify that the functionality requires  $v_f = 1$ . Each functionality may introduce several requirements, i.e., several essential sets of states. For instance, a “move” functionality could have two separate requirements, one for cruise motion and one for slow motion.

The list of `req` predicates can then be extracted to produce permissiveness properties of the form:  $\text{AG}(\text{EF}(\text{req}))$ . We choose to reuse the same template as the one for universal permissiveness. However, we now require the reachability of sets of states, rather than the reachability of every individual state. For example, we have no reachability requirement on states satisfying  $v_f = 2$ , and may accommodate strategies discarding some of the states  $v_f = 1$  provided that at least one of them remains reachable.

The functionalities that would require another type of template are not considered yet, but so far the expressivity of this template has been sufficient to model the considered functionalities.

## 4.2 Binding Together Invariants and Permissiveness

The permissiveness and the safety properties are defined using two different state models. Some of the abstract variables used in those state models represent the same physical observation, or dependent ones. To connect the invariants and functionalities models, we have to bind their variables. Two types of bindings can be used: physical dependencies (e.g., speed and acceleration), or the use of the same observation with two different partitions.

In the first case, we specify the constraints on transitions (using the NuSMV keyword `TRANS`) or on states (`INVAR`). For example, for observations of speed and accelerations, we would write `TRANS next(acc) = 0  $\rightarrow$  next(speed) = speed`. The NuSMV primitives `next(acc)` or `next(speed)` specify the value of `acc` or `speed` after transitioning, i.e., if the acceleration is null, the speed cannot change.

In the second case, we need to introduce a “glue” variable to bind the different partitions. This variable will be partitioned in as many intervals as needed. The different intervals will be bound with a specification on the states. For

example, let us assume we have an invariant and a functionality using a velocity variable, and the partition used for the invariant is  $v_{inv} = \{0, 1, 2\}$  where 0 : stationary or slow, 1 : medium and 2 : high, and the one used for the functionality is  $v_f = \{0, 1\}$  where 0 : stationary or slow and 1 : medium or high. We introduce a continuous “glue” variable partitioned as  $v_{glue} = \{0, 1, 2\}$ . The binding through the “glue” variable is specified as follows:

INVAR  $v_{glue} = 0 \leftrightarrow v_{inv} = 0 \ \& \ v_f = 0$ ;  
 INVAR  $v_{glue} = 1 \leftrightarrow v_{inv} = 1 \ \& \ v_f = 1$ ;  
 INVAR  $v_{glue} = 2 \leftrightarrow v_{inv} = 2 \ \& \ v_f = 1$ .

Note that those two binding approaches, by adding constraints or glue variables, are also used in the standard SMOF process when merging models of different safety invariants.

### 4.3 Restricting Functionalities

Custom permissiveness is weaker than SMOF’s generic permissiveness, since we get rid of non essential reachability requirements. As a result, the strategy synthesis tool may return relevant strategies that would have been discarded with the generic version.

Still, it is possible that the custom requirements do not suffice, and that no strategy is found by the tool. We want to make it possible that the user restricts the functionalities, i.e., further weakens permissiveness. This may change the system’s objectives, or increase the time or effort it takes to achieve the objectives. Functionalities can be restricted in several ways. We consider three of them in this paper. An important point is that the corresponding trade-offs are made explicit in the model of functionalities.

First, one of the  $\text{req}$  predicates can be weakened to correspond to a larger set of system states. The permissiveness property becomes  $\text{AG}(\text{EF}(\text{req}'))$ , with  $\text{req} \Rightarrow \text{req}'$ . For example, the “move” functionality can be restricted to “move slowly”, where  $\text{req}'$  means that  $V$  only needs to reach a velocity  $V_{\text{req}'}$  lower than the initially specified cruise one.

Second, it is possible to replace the universal permissiveness property by the simple one,  $\text{EF}(\text{req})$ . This weak form of permissiveness was already offered by the generic version of SMOF, but it applied to all individual states. With the custom list of  $\text{req}$  predicates, the user can decide for which of these predicates simple permissiveness would be acceptable. For example, the “move” functionality could become impossible after some monitor’s intervention has been triggered, but other functionalities like manipulating objects would have to be preserved.

The third way is to simply remove the functionality from the requirements. For example a “manipulation while moving” functionality is no longer required. Here, the corresponding CTL property is simply deleted, and the synthesis run again without it. This ultimate restriction step can show the user that the intended functionality is not compatible with the required level of safety. This information can be propagated back to the hazard analysis step and used to revise the design of the system or its operation rules. Again, not all of the



requirements may need a restriction. The functionalities that are not restricted are guaranteed to remain fully available despite the monitor’s intervention.

#### 4.4 Integration in SMOF Tooling

Integrating management of custom permissiveness into SMOF is possible without any major change of the core toolset. Only the front-end would need to be updated. The template would include an optional functionality modeling part with syntactic sugar to introduce the list of `req` predicates. The auto-completion facilities would allow for the generation of the CTL properties from the `req` predicates and the desired level of permissiveness (universal, simple) for each of them. The core modeling approach and strategy synthesis algorithm would remain unchanged.

The SMOF front-end has not been updated yet. Still, we can use the current version of SMOF for the experimentation, as presented in the next Section. We manually entered the CTL properties (rather than just the `req` predicates) and intercepted a generated command script to instruct the synthesis tool to use the custom properties, not the generic ones.

## 5 Application to an Example

To explain the approach and study its usefulness we will consider a robotic system composed of a mobile platform and an articulated arm (see Fig. 2). It is an industrial co-worker in a manufacturing setting, sharing its workspace with human workers. Its purpose is to pick up objects using its arm and to transport them. To do so, the arm can rotate and a gripper at its end can open and close to hold objects. Some areas of the workspace are prohibited to the robot. The hazard analysis has been performed on this robot in a previous project [20] and a list of 13 safety invariants has been identified. Among them, 7 could be handled by the monitor [15]. Several ways to model those invariants have been explored, considering various interventions and observation means. We defined custom permissiveness properties for each of the models and detail here the three invariants that give different results compared to generic permissiveness:

- $SI_1$ : the arm must not be extended when the platform moves with a speed higher than  $s_{max}$ ;
- $SI_2$ : a gripped box must not be tilted more than  $\alpha_0$ ;
- $SI_3$ : the robot must not enter a prohibited zone.

The models for the three mentioned invariants can be found online at [2]. For all of them, the synthesis took less than 0.5s to compute on an Intel Core i5-3437U CPU @ 1.90 GHz x 4 with 16 GB of memory.

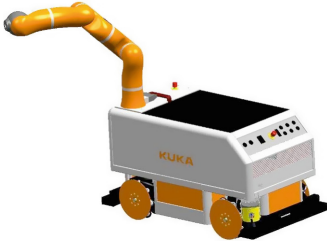


Fig. 2. Manipulator robot from Kuka

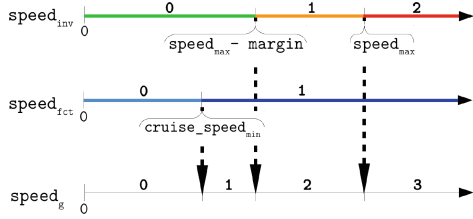


Fig. 3. Partitioning of the  $s_g$  variable

### 5.1 $SI_1$ : The Arm Must Not Be Extended when the Platform Moves over a Certain Speed

**Modeling.** We consider the invariant  $SI_1$ : *the arm must not be extended when the platform moves with a speed higher than  $s_{max}$* . The available observations are  $s_{inv}$ , the speed of the platform; and  $a_{inv}$ , the position of the arm. Note that the variables names are extended with the index  $inv$  to specify that they are the variables used for the invariant model. The observations are partitioned as detailed in Table 1. Considering the discrete representation of the variables, the catastrophic state can be expressed as  $cata : s_{inv} = 2 \ \& \ a_{inv} = 1$  (high speed with extended arm).

Table 1. Partitioning of the variables  $s_{inv}$  and  $a_{inv}$

Speed of the platform	Real speed interval	Discrete variable
Low	$s_{inv} < s_{max} - m$	$s_{inv} = 0$
Within the margin	$s_{max} - m \leq s_{inv} < s_{max}$	$s_{inv} = 1$
Higher than the maximum allowed value	$s_{inv} \geq s_{max}$	$s_{inv} = 2$
Position of the arm		Discrete variable
Not extended beyond the platform		$a_{inv} = 0$
Extended beyond the platform		$a_{inv} = 1$

To express the relevant permissiveness properties, we identify in the specifications what functionalities are related to the invariant. Let us consider the variables involved in  $SI_1$ . The  $s_{inv}$  variable is an observation of the speed of the mobile platform, in absolute value. The system is supposed to move around the workplace to carry objects, i.e., the speed must be allowed to reach a minimal cruise speed value  $c\_s_{min}$ , from any state. To model this functionality we introduce the  $s_{fct}$  variable, which will be partitioned as showed in Table 2. Note that the variables names are extended with the index  $fct$  to specify that they are the variables used for the functionalities model. This property can be expressed following the template:  $cruise\ motion : AG(EF(s_{fct} = 1))$ .

**Table 2.** Partitioning of the variables  $\mathbf{s}_{fct}$  and  $\mathbf{a}_{fct}$ 

Speed of the platform	Real speed interval	Discrete variable
Null or lower than the minimum cruise speed	$s_{fct} < c_{-}s_{min}$	$\mathbf{s}_{fct} = 0$
At the minimum cruise speed or higher	$s_{fct} \geq c_{-}s_{min}$	$\mathbf{s}_{fct} = 1$
Position of the arm		Discrete variable
Folded		$\mathbf{a}_{fct} = 0$
Extended beyond the platform		$\mathbf{a}_{fct} = 1$

The system must also be able to stop or move slowly, thus another functionality is expressed: *slow motion* :  $\text{AG}(\text{EF}(s_{fct} = 0))$ . Also, the  $\mathbf{a}_{inv}$  variable models whether the manipulator arm is extended beyond the platform or not. To handle objects, the arm must be allowed from any state to reach a state where the arm is extended beyond the platform, and a state where the arm is folded. We introduce the variable  $\mathbf{a}_{fct}$  which is partitioned as showed in Table 2. We have *arm extension* :  $\text{AG}(\text{EF}(a_{fct} = 1))$  and *arm folding* :  $\text{AG}(\text{EF}(a_{fct} = 0))$ .

The speed value and arm position are observed in both the invariant model ( $\mathbf{s}_{inv}$  and  $\mathbf{a}_{inv}$ ) and the functionalities model ( $\mathbf{s}_{fct}$  and  $\mathbf{a}_{fct}$ ). We need to make their evolution consistent. To do so, we introduce glue variables,  $\mathbf{s}_g$  and  $\mathbf{a}_g$ .

For the speed, we have two different partitions as presented in Fig. 3, one for  $\mathbf{s}_{inv}$  (with discrete values  $\{0, 1, 2\}$ ) and one for  $\mathbf{s}_{fct}$  (with discrete values  $\{0, 1\}$ ). The resulting glue variable  $\mathbf{s}_g$  will then have four values as presented in Fig. 3. We thus have the formal definition:

$$\begin{aligned} \text{INVAR } \mathbf{s}_g = 0 &\leftrightarrow \mathbf{s}_{inv} = 0 \ \& \ \mathbf{s}_{fct} = 0; \\ \text{INVAR } \mathbf{s}_g = 1 &\leftrightarrow \mathbf{s}_{inv} = 0 \ \& \ \mathbf{s}_{fct} = 1; \\ \text{INVAR } \mathbf{s}_g = 2 &\leftrightarrow \mathbf{s}_{inv} = 1 \ \& \ \mathbf{s}_{fct} = 1; \\ \text{INVAR } \mathbf{s}_g = 3 &\leftrightarrow \mathbf{s}_{inv} = 2 \ \& \ \mathbf{s}_{fct} = 1. \end{aligned}$$

For the arm variable, it is much simpler:

$$\begin{aligned} \text{INVAR } \mathbf{a}_g = 0 &\leftrightarrow \mathbf{a}_{inv} = 0 \ \& \ \mathbf{a}_{fct} = 0; \\ \text{INVAR } \mathbf{a}_g = 1 &\leftrightarrow \mathbf{a}_{inv} = 1 \ \& \ \mathbf{a}_{fct} = 1. \end{aligned}$$

Additionally, we have to provide a model of the interventions of the monitor. Let us consider that two interventions are available: the brakes can be triggered and affect the speed, and the extension of the arm can be blocked. Concerning the braking intervention, it can be applied at any time but will only be efficient (i.e. prevent the reachability of  $\mathbf{s}_{inv} > \mathbf{s}_{max}$ ) if it is engaged when the speed threshold  $\mathbf{s}_{max} - m$  has just been crossed. Indeed, the size of the margin is chosen precisely to have time to brake before reaching the undesired value. For the intervention blocking the arm, its effect is to block the extension and it can only be applied if the arm is not already extended (see definitions in Table 3).

**Results.** We compare in this section the results obtained without and with the approach through the definition of custom permissiveness. To graphically

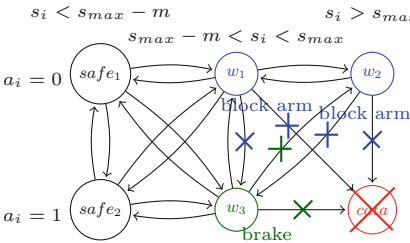
**Table 3.** Interventions definition for  $SI_1$

Name	Precondition	Effect
Brake	$s_{inv} = 0 \ \& \ next(s_{inv}) = 1$	$next(s_{inv}) = s_{inv} - 1$
Block_arm	$a_{inv} = 0$	$next(a_{inv}) = 0$

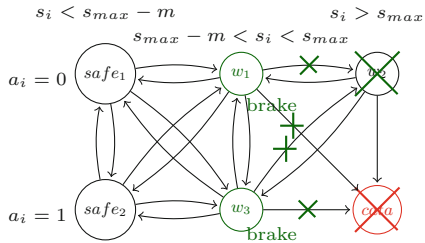
**Table 4.** Interventions definition for  $SI_2$

Name	Precondition	Effect
Brake	$d = 2 \ \& \ next(d) = 1$	$next(v_{inv}) = 0$

describe the strategies, we represent the invariant as a state machine. In the first case, we use the generic permissiveness, i.e., the reachability of every non-catastrophic state (the states  $\{safe_1, safe_2, w_1, w_2, w_3\}$  in Fig. 4), from every other non-catastrophic state. Only one minimal strategy (no useless interventions) is both safe and permissive. It is found by SMOF and represented in Fig. 4.



**Fig. 4.** Single strategy synthesized for the invariant  $SI_1$  with generic permissiveness properties.



**Fig. 5.** Additional strategy synthesized for the invariant  $SI_1$  with the custom permissiveness properties.

In the second case, we replace the generic permissiveness with the use of the custom permissiveness properties cruise motion, slow motion, arm folding and arm extension specified before. We only require the reachability of the states  $\{safe_1, safe_2\}$ . After running the synthesis, in addition to the previous strategy we have a strategy only using the braking intervention (see Fig. 5). This can be preferable in some cases, as the use of the arm is then never impacted and even if the monitor triggers the brakes the system can keep manipulating objects. This strategy couldn't be found with the generic permissiveness as it removes the reachability of  $w_2$ .

The custom permissiveness requirements may allow to synthesize more strategies, like in the previous example, or even to synthesize strategies for problems that had no solution with the generic permissiveness. Indeed, we require the reachability of a reduced set of states, therefore more strategies can be found.

## 5.2 $SI_2$ : A Gripped Box Must Not Be Tilted More Than $\alpha_0$

For the initial model presented in [15], the monitor could observe the presence of a box (inferred through the position of the robot’s arm in the workspace and the position of the gripper), and the angle of rotation of the arm. No strategy was found. Indeed, the monitor only could brake the arm (prevent its rotation) and no control was possible on the gripper. The monitor would thus not be able to prevent the system from grabbing a box with the gripper already tilted more than  $\alpha_0$ . We chose to reformulate the invariant as  $SI_2$ : *a gripped box must not be tilted more than  $\alpha_0$  if the robot is outside of the storage area*. The industrial partner indicated that dropping a box (tilt it over  $\alpha_0$ ) in the storage area is not that dangerous as it would fall from a low height.

As an alternative solution to ensure  $SI_2$ , we also explored the effect of an additional intervention: the monitor can lock the gripper (prevent it from closing). But the automated synthesis with the generic permissiveness failed to return any strategy. We now revisit this model by specifying custom permissiveness properties for a manipulation functionality (carrying a box at a low rotation angle). Using the custom permissiveness, the tool successfully synthesizes a strategy. It combines the braking of the arm and the lock of the gripper to maintain the invariant while permitting functionality.

## 5.3 $SI_3$ : The Robot Must Not Enter a Prohibited Zone

**Modeling.** The considered invariant is:  $SI_3$ : *the robot must not enter a prohibited zone*. The observation used is  $d$ , the distance to the prohibited zone. The distance variable is partitioned according to the concept of margin:  $\mathbf{d} : \{0, 1, 2\}$ , 0 representing the robot into the prohibited zone, 1 the robot close to the prohibited zone and 2 the robot far from the prohibited zone. According to this partition, the catastrophic state can be expressed as  $\mathbf{cata} : \mathbf{d} = 0$ .

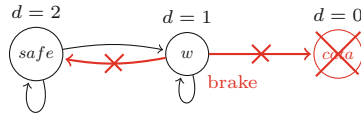
The only available intervention here is the braking intervention, which stops the robot completely. To model this intervention, we introduce a velocity variable  $\mathbf{v}_{\text{inv}}$ , partitioned as follows:  $\mathbf{v}_{\text{inv}} : \{0, 1\}$  where 0 represents the robot stopped and 1 the robot moving. A dependency between the distance and the velocity variables is specified as  $\text{TRANS next}(\mathbf{v}_{\text{inv}}) = 0 \rightarrow \text{next}(\mathbf{d}) = \mathbf{d}$ , i.e., the distance cannot change if the robot does not move. The braking intervention is only effective under the precondition that the distance threshold to the prohibited zone has just been crossed, and affects the velocity variable. This intervention is modeled as shown in Table 4.

In this case, for the functionalities we just need to specify that the robot needs to reach a state where it is moving, and a state where it is stopped. We model the custom permissiveness with  $\text{move} : \text{AG}(\text{EF}(\mathbf{v}_{\text{fct}} = 1))$  and  $\text{stop} : \text{AG}(\text{EF}(\mathbf{v}_{\text{fct}} = 0))$  where  $\mathbf{v}_{\text{fct}}$  represents the robot moving or stopped. This variable is directly bound to the  $\mathbf{v}_{\text{inv}}$  variable with a glue variable  $\mathbf{v}_{\mathbf{g}}$  as:

$$\begin{aligned} \text{INVAR } \mathbf{v}_{\mathbf{g}} = 0 &\leftrightarrow \mathbf{v}_{\text{inv}} = 0 \ \& \ \mathbf{v}_{\text{fct}} = 0; \\ \text{INVAR } \mathbf{v}_{\mathbf{g}} = 1 &\leftrightarrow \mathbf{v}_{\text{inv}} = 1 \ \& \ \mathbf{v}_{\text{fct}} = 1. \end{aligned}$$

**Results.** The synthesis of strategies with the braking intervention and the move and stop functionalities does not give any result. Applying the brakes until the system is stopped violates the permissiveness property associated to the move functionality. The system is stopped close to the prohibited zone and cannot ever move again, i.e., the monitor’s intervention is irreversible. In term of automata, we reached a deadlock state. In order to guarantee safety and synthesize a strategy, we need to either change the interventions or accept a restriction of the functionality as described in Sect. 4.3. In our case, we do not have any other intervention, we thus need to restrict the functionality.

We can express the restriction as follows: we accept the intervention of the monitor to be irreversible, but we still want the functionality to be fully available before the intervention of the monitor. We have the following resulting permissiveness property: restricted move :  $EF(v_{\text{fct}} = 1)$ . The property for stop remains unchanged. With the restricted permissiveness property, the synthesis generates one strategy which is modeled in Fig. 6.



**Fig. 6.** Strategy synthesized for the invariant  $SI_3$  with restricted functionality.

As we can see, the  $w$  state is a deadlock: the system cannot reach any other state from this state. It means that if the robot ever gets too close to a prohibited zone, it will be stopped by the monitor and an intervention of the operator will be needed to continue the mission. The safety is guaranteed by this strategy. Specifying the restriction of functionalities highlights the impact of the monitor intervention on the system ability to function, which was not possible with the use of generic simple permissiveness.

## 6 Conclusion and Perspectives

In this paper, we have described an approach to specify safety monitors for robots, using and extending the SMOF monitoring framework. We overcome an overly stringent definition of the monitor’s permissiveness in proposing a custom definition of permissiveness according to the system’s functionalities (the behaviors necessary to fulfill its purposes). The custom permissiveness properties are expressed following a simple template. We require the reachability of a reduced set of states, therefore, more strategies can be synthesized. In the studied example, the proposed solution provided a new strategy only requiring the use of one intervention instead of two. Also, a problem which had no solutions with a generic definition of permissiveness properties had one with custom properties.

Whenever it is not possible to synthesize a safety strategy, we propose an iterative design strategy: we give three ways to adapt functionalities by weakening the permissiveness properties following a template. In these situations, some strategies can often still be found with slight and traceable changes of the functionalities. The impact of the monitor on the robot's operation can thus be qualified and reasoned about.

Integrating the definition and use of custom permissiveness properties is now possible with the existing SMOF tooling with a small change on the front-end. The synthesis algorithm remains unchanged. In future work we wish to adapt the template so that the user does not have to use the CTL logic.

We would also like to extend our approach to cover different types of monitor interventions. We could search for multi-level strategies combining guaranteed and non-guaranteed interventions (having a probability of success, possibly depending on the operational situation). The monitor would first try the interventions that would affect the system without compromising the mission (e.g., trajectory re-planning). In case of failure of those interventions, the least permissive but guaranteed ones (e.g., emergency stop) would only be triggered in last emergency.

## References



1. NuSMV home page. <http://nusmv.fbk.eu/>. Accessed Nov 2017
2. Safety Monitoring Framework. LAAS-CNRS Project. <https://www.laas.fr/projects/smf>. Accessed Dec 2017
3. Adam, S., Larsen, M., Jensen, K., Schultz, U.P.: Rule-based dynamic safety monitoring for mobile robots. *J. Softw. Eng. Robot.* **7**, 120–141 (2016)
4. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.* **1**, 11–33 (2004)
5. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. *Trans. Softw. Eng.* **30**, 859–872 (2004)
6. Falcone, Y., Fernandez, J.-C., Mounier, L.: What can you verify and enforce at runtime? *Int. J. Softw. Tools Technol. Transf.* **14**, 349–382 (2012)
7. Fox, J., Das, S.: *Safe and Sound - Artificial Intelligence in Hazardous Applications*. AAAI Press/MIT Press, Palo Alto (2000)
8. Guiochet, J.: Hazard analysis of human-robot interactions with HAZOP-UML. *Saf. Sci.* **84**, 225–237 (2016)
9. Haddadin, S., Suppa, M., Fuchs, S., Bodenmüller, T., Albu-Schäffer, A., Hirzinger, G.: Towards the robotic co-worker. In: Pradalier, C., Siegwart, R., Hirzinger, G. (eds.) *The 14th International Symposium on Robotics Research (ISRR2011)*, vol. 70, pp. 261–282. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-19457-3\\_16](https://doi.org/10.1007/978-3-642-19457-3_16)
10. Huang, J., Erdogan, C., Zhang, Y., Moore, B., Luo, Q., Sundaresan, A., Rosu, G.: ROSRV: runtime verification for robots. In: Bonakdarpour, B., Smolka, S.A. (eds.) *RV 2014. LNCS*, vol. 8734, pp. 247–254. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-11164-3\\_20](https://doi.org/10.1007/978-3-319-11164-3_20)
11. Jiang, H., Elbaum, S., Detweiler, C.: Inferring and monitoring invariants in robotic systems. *Auton. Robot* **41**, 1027–1046 (2017)

12. Leucker, M., Schallhart, C.: A brief account of runtime verification. *J. Log. Algebr. Program.* **78**, 293–303 (2009)
13. Ligatti, J., Bauer, L., Walker, D.: Edit automata: enforcement mechanisms for run-time security policies. *IJIS* **4**, 2–16 (2005)
14. Machin, M., Dufossé, F., Blanquart, J.-P., Guiochet, J., Powell, D., Waeselynck, H.: Specifying safety monitors for autonomous systems using model-checking. In: Bon-davalli, 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](https://doi.org/10.1007/978-3-319-10506-2_18)
15. Machin, M., Guiochet, J., Waeselynck, H., Blanquart, J.-P., Roy, M., Masson, L.: SMOF - a safety monitoring framework for autonomous systems. *IEEE Trans. Syst., Man Cybern.* **PP**, 1–14 (2016)
16. Martinelli, F., Matteucci, I., Morisset, C.: From qualitative to quantitative enforcement of security policy. In: Kotenko, I., Skormin, V. (eds.) *MMM-ACNS 2012*. LNCS, vol. 7531, pp. 22–35. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-33704-8\\_3](https://doi.org/10.1007/978-3-642-33704-8_3)
17. Masson, L., Guiochet, J., Waeselynck, H., Desfosses, A., Laval, M.: Synthesis of safety rules for active monitoring: application to an airport light measurement robot. In: *2017 First IEEE International Conference on Robotic Computing (IRC)* (2017)
18. Pace, C., Seward, D.: A safety integrated architecture for an autonomous safety excavator. In: *International Symposium on Automation and Robotics in Construction* (2000)
19. Roderick, S., Roberts, B., Atkins, E., Akin, D.: The ranger robotic satellite servicer and its autonomous software-based safety system. *Intell. Syst.* **19**, 12–19 (2004)
20. SAPHARI: Safe and Autonomous Physical Human-Aware Robot Interaction. Project supported by the European Commission under the 7th Framework Programme. (2011–2015). [www.saphari.eu](http://www.saphari.eu). Accessed Nov 2017
21. Woodman, R., Winfield, A.F., Harper, C., Fraser, M.: Building safer robots: safety driven control. *Int. J. Robot. Res.* **31**, 1603–1626 (2012)





# Sound Black-Box Checking in the LearnLib

Jeroen Meijer<sup>(✉)</sup>  and Jaco van de Pol 

Formal Methods and Tools, University of Twente, Enschede, The Netherlands  
{j.j.g.meijer,j.c.vandepol}@utwente.nl

**Abstract.** In Black-Box Checking (BBC) incremental hypotheses of a system are learned in the form of finite automata. On these automata LTL formulae are verified, or their counterexamples validated on the actual system. We extend the LearnLib’s system-under-learning API for sound BBC, by means of state equivalence, that contrasts the original proposal where an upper-bound on the number of states in the system is assumed. We will show how LearnLib’s new BBC algorithms can be used in practice, as well as how one could experiment with different model checkers and BBC algorithms. Using the RERS 2017 challenge we provide experimental results on the performance of all LearnLib’s active learning algorithms when applied in a BBC setting. The performance of learning algorithms was unknown for this setting. We will show that the novel incremental algorithms TTT, and ADT perform the best.

## 1 Introduction

There are many formal methods for analyzing the desired behavior of systems. Examples include complex industrial critical systems, such as wafer steppers, and X-ray diffraction machines. In these systems both liveness (something good eventually happens), and safety (something bad never happens) are essential. It is key for testers and developers of these systems to have easily usable tooling available to investigate liveness and safety properties of systems. We present an instance of such tooling known as Black-Box Checking (BBC), originally developed by Peled et al. [23] which we implemented in the LearnLib. We show its ease of use, why our method is sound even when not assuming an upper-bound on the number of states in the System Under Learning (SUL), and show how well it performs with an actual case study.

The essence of using formal methods is relating requirements on one hand, and a system on the other. The requirements are often formulated with some kind of temporal logic, such as Linear Temporal Logic (LTL). These formulas then express the liveness and safety properties of the system. In formal methods traditionally, the three main complementary methods are *verification*, *testing*, and *learning*. Verification involves checking whether some abstract instance (e.g.

---

J. Meijer—Supported by STW SUMBAT grant: 13859.

J. van de Pol—Supported by the 3TU.BSR project.

in the form of an automaton) of the specification adheres to a set of requirements. Testing involves checking whether the system conforms to an abstract instance of the specification. If such an abstract instance is modeled as an automaton, Model-Based Testing (MBT) [30] is typically applied. Conversely, an abstract instance can also be learned from a system. If such an instance is in the form of an automaton, and the system can only be accessed as a black-box, then this procedure is called Active Automata Learning (AAL) [27]. LearnLib [12] is a toolset that contains a wide variety of AAL algorithms. Many of these algorithms are inspired by Angluin’s famous  $L^*$  algorithm [1]. Figure 1 provides an overview of the aforementioned approaches. Figure 1 also shows the concept of an *alphabet*. An alphabet contains the symbols in which requirements must be written, and in what language the system communicates with the environment. This means that to make the system perform an action an input must be sent that is a symbol in the alphabet. To observe the reaction of the system, the output must also be a symbol in the alphabet.

Testing, verification, and learning can be used in a complementary fashion, because all of them have their advantages. Verification is typically done through model checking. Model checking has been around for several decades and efficient model checkers are readily available. The advantage of testing is a highly automated approach to check whether a system conforms to a specific model. There are many mature MBT tools available, such as JTorX [3]. From a practical perspective, learning an automaton from a system is also quite straightforward, because the only requirements are a definition of the alphabet, and some kind of adapter between a learning algorithm and system. These adapters are often quite easy to build. The three methods also have disadvantages. For example when verification is performed, it is known which requirements hold on an abstract notion of the system, but it is unknown which of those requirements also hold on the actual system. Testing has the disadvantage that the abstract notion (e.g. an automaton) has to be built and maintained by hand. Writing specifications for automata can be tedious, since it is often done with specification languages that may be unfamiliar to the developers of the system. Verifying requirements on an automaton that is obtained through learning is also difficult. Because it can take quite a long time before learning algorithms produce such an automaton. Even when such an automaton is obtained, verifying requirements is not straightforward, because the learned automaton can be incorrect. Black-box checking tries to alleviate those problems. It resolves the need for maintenance of an abstract notion of a system so that requirements can be directly checked on a system.

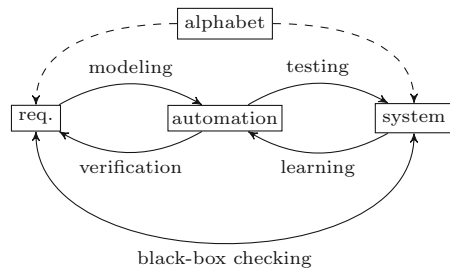


Fig. 1. Formal methods

These adapters are often quite easy to build. The three methods also have disadvantages. For example when verification is performed, it is known which requirements hold on an abstract notion of the system, but it is unknown which of those requirements also hold on the actual system. Testing has the disadvantage that the abstract notion (e.g. an automaton) has to be built and maintained by hand. Writing specifications for automata can be tedious, since it is often done with specification languages that may be unfamiliar to the developers of the system. Verifying requirements on an automaton that is obtained through learning is also difficult. Because it can take quite a long time before learning algorithms produce such an automaton. Even when such an automaton is obtained, verifying requirements is not straightforward, because the learned automaton can be incorrect. Black-box checking tries to alleviate those problems. It resolves the need for maintenance of an abstract notion of a system so that requirements can be directly checked on a system.

When BBC is applied to industrial cases, the guess of an upper-bound on the number of states to have a sound BBC procedure can be either dangerous (the guess is too low), or unpractical (the guess is too high). We resolve this by allowing the LearnLib to check for state equivalence in the SUL. Our implementation in the LearnLib is Free and Open Source, this alleviates the current scarcity of tool support. To investigate *how efficient several active learning algorithms are for BBC*, we contribute the following.

- Two variations of black-box checking algorithms.
- A novel sound black-box checking approach that uses state equivalences, instead of an upper-bound on the number of states in the SUL.
- A modular design, allowing new model checkers to be added easily, or smarter strategies to be implemented for detecting spurious counterexamples.
- A thorough reproducible experimental setup, with several algorithms.

The rest of the paper is structured as follows. Section 2 provides preliminary definitions and procedures for model checking, active learning and black-box checking. Section 3 describes how one can check whether a SUL accepts an infinite lasso-shaped word, and how this is implemented in the LearnLib. In Sect. 4 we discuss related work, such as other model checkers, active learning algorithms and the LBTest toolset. Section 5 details the result of our case study, and Sect. 6 concludes our work.

## 2 Preliminaries

The LearnLib mainly contains AAL algorithms for DFAs and Mealy machines. We provide a definition for both, and a definition for LTSs where multiple labels per edge are allowed. Typically, model checkers, such as LTSmin verify LTL properties on LTSs. Hence we provide LTL semantics for LTSs, and provide straightforward translations from DFAs and Mealy machines to LTSs. We also provide actual implementations of these translations in the LearnLib. Furthermore, this section gives a short introduction to active learning, and black-box checking.

**Definition 1 (Edge Labeled Transition System).** *An edge Labeled Transition System (LTS) is defined as a tuple  $\mathcal{L} = \langle S, s_0, \delta, L, T, \lambda \rangle$ , where  $S$  is a finite nonempty set of states,  $s_0 \in S$  is the initial state,  $\delta : S \rightarrow 2^S$  is the transition function,  $L$  is the set of edge labels,  $T$  is the set of edge label types, and  $\lambda : S \times S \rightarrow 2^{T \times L}$ : is the edge labeling function. A path in  $\mathcal{L}$  is an infinite sequence of states beginning in  $s_0$ . The set of paths is  $\text{Paths}(\mathcal{L}) = \{s_0 s_1 \dots \in S^\omega \mid \forall i > 0: s_i \in \delta(s_{i-1})\}$ . A trace is an infinite sequence of sets of tuples of labels:  $\text{Traces}(\mathcal{L}) = \{\lambda(s_0, s_1)\lambda(s_1, s_2) \dots \in (2^{T \times L})^\omega \mid s \in \text{Paths}(\mathcal{L})\}$ .*

**Definition 2 (Deterministic Finite Automaton).** *A Deterministic Finite Automaton (DFA) is defined as a tuple  $\mathcal{D} = \langle S, s_0, \Sigma, \delta, F \rangle$ , where  $S$  is a finite nonempty set of states,  $s_0 \in S$  is the initial state,  $\Sigma$  is a finite alphabet,  $\delta : S \times \Sigma \rightarrow S$  is the total transition function,  $F \subseteq S$  is the set*

of accepting states. The language of  $\mathcal{D}$  is denoted  $L(\mathcal{D})$ . A DFA is Prefix-Closed iff  $\forall s \in S, \forall i \in \Sigma: \delta(s, i) \in F \implies s \in F$ . In other words  $\forall \sigma_1 \dots \sigma_n \in L(\mathcal{D}): \sigma_1 \dots \sigma_{n-1} \in L(\mathcal{D})$ . The LTS of a non-empty, prefix-closed DFA  $\mathcal{D}$  is  $\mathcal{L}_{\mathcal{D}} = \langle F, s_0, \delta_{\mathcal{L}}, \Sigma, \{\text{letter}\}, \lambda_{\mathcal{L}} \rangle$ , where  $\delta_{\mathcal{L}}(s) = \bigcup_{i \in \Sigma} \delta(s, i)$ , and  $\lambda_{\mathcal{L}}(s, s') = \{(\text{letter}, l) \mid l \in \Sigma \wedge \delta(s, l) = s'\}$ .



Fig. 2. Example DFA

Example 1 (DFA). An example prefix-closed DFA for the regular expression  $(ab)^*a?$  is given in Fig. 2a (the trap state is implicit). The LTS is given in Fig. 2b. The traces in the LTS are:  $\{ \{(\text{letter}, a)\} \{(\text{letter}, b)\} \dots \}$ .

Definition 3 (Mealy Machine). A Mealy machine is defined as a tuple  $\mathcal{M} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$ , where  $S$  is a finite nonempty set of states,  $s_0 \in S$  is the initial state,  $\Sigma$  is a finite input alphabet,  $\Omega$  is a finite output alphabet,  $\delta: S \times \Sigma \rightarrow S$  is the total transition function, and  $\lambda: S \times \Sigma \rightarrow \Omega$  is the total output function. The LTS of  $\mathcal{M}$  is  $\mathcal{L}_{\mathcal{M}} = \langle S, s_0, \delta_{\mathcal{L}}, \Sigma \cup \Omega, \{\text{input}, \text{output}\}, \lambda_{\mathcal{L}} \rangle$ , where  $\delta_{\mathcal{L}}(s) = \bigcup_{i \in \Sigma} \delta(s, i)$ , and  $\lambda_{\mathcal{L}}(s, s') = \{ \{(\text{in}, i), (\text{out}, o)\} \mid i \in \Sigma \wedge \delta(s, i) = s' \wedge o \in \Omega \wedge \lambda(s, i) = o \}$ .

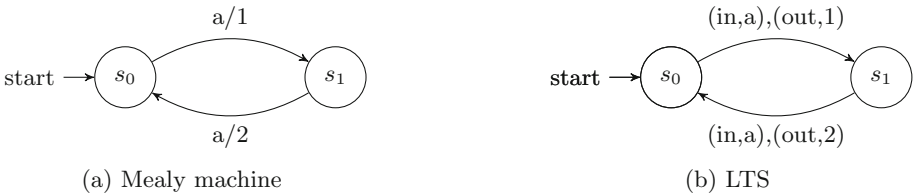


Fig. 3. Example Mealy machine

Example 2 (Mealy Machine). An example Mealy machine is given in Fig. 3a. The LTS is given in Fig. 3b. The traces of the LTS are:  $\{ \{(\text{in}, a), (\text{out}, 1)\} \{(\text{in}, a), (\text{out}, 2)\} \dots \}$ .

Throughout this paper the following assumptions are made.

- All DFAs reject the empty language (because an LTS thereof is not defined).
- All DFAs are prefix-closed (Mealy machines are by definition prefix-closed).
- All DFAs and Mealy machines are minimal (automata constructed through active learning are always minimal; our definition of prefix-closed only holds on minimal automata).
- All SULs are deterministic.

## 2.1 LTL Model Checking

An LTL formula expresses a property that should hold over all infinite runs of a system. This means that if a system does not satisfy an LTL property, there generally exists a counterexample that is an infinite word which exhibits a lasso structure.

**Definition 4 (LTL).** *Given an LTS  $\mathcal{L} = \langle S, s_0, \delta, L, T, \lambda \rangle$ , LTL formulae over  $\mathcal{L}$  adhere to the following grammar:<sup>1</sup>  $\phi ::= \mathbf{true} \mid \phi_1 \wedge \phi_2 \mid \neg \phi \mid \mathbf{X} \phi \mid \phi_1 \mathbf{U} \phi_2 \mid t = l$ , where  $t \in T$ , and  $l \in L$ . Given an LTL formula  $\phi$ , all infinite words that satisfy  $\phi$  are given by the set  $\text{Words}(\phi) = \{\sigma \in (2^{T \times L})^\omega \mid \sigma \models \phi\}$ , where the satisfaction relation  $\models \subseteq (T \times L)^\omega \times \text{LTL}$  is defined inductively over  $\phi$  by the following properties. Let  $\sigma = A_0 A_1 A_2 \dots \in (2^{T \times L})^\omega$ , and  $\sigma[j \dots] = A_j A_{j+1} A_{j+2} \dots$ :*

$$\sigma \models \begin{cases} \mathbf{true}, \\ \phi_1 \wedge \phi_2 & \text{iff } \sigma \models \phi_1 \wedge \sigma \models \phi_2, \\ \neg \phi & \text{iff } \sigma \not\models \phi, \\ \mathbf{X} \phi & \text{iff } \sigma[1 \dots] = A_1 A_2 A_3 \dots \models \phi, \\ \phi_1 \mathbf{U} \phi_2 & \text{iff } \exists j \geq 0: \sigma[j \dots] \models \phi_2 \wedge \forall 0 \leq i < j: \sigma[i \dots] \models \phi_1, \\ t = l & \text{iff } (t, l) \in A_0. \end{cases}$$

Finally  $\mathcal{L} \models \phi \iff \text{Traces}(\mathcal{L}) \subseteq \text{Words}(\phi)$ .

*Example 3 (LTL for DFAs).* An example LTL formula that holds for the LTS  $\mathcal{L}$  in Fig. 2b is:  $\phi = \mathbf{X}(\text{letter} = b)$ . All the words that satisfy the formula are in  $\text{Words}(\phi) = \{\{(letter, a)\}\{(letter, b)\} \dots, \{(letter, b)\}\{(letter, b)\} \dots\}$ . Clearly,  $\text{Traces}(\mathcal{L}) \subseteq \text{Words}(\phi)$ , so  $\mathcal{L} \models \phi$ .

An example for Mealy machines is analogous. Finally we provide a formal definition of a lasso as follows.

**Definition 5 (Lasso).** *Given an LTS  $\mathcal{L}$ , a trace  $\sigma \in \text{Traces}(\mathcal{L})$  is a lasso if it can be split in a finite prefix  $p$ , such that  $p \sqsubset \sigma$ , and a finite loop  $q$ , such that  $pq^\omega = \sigma$ .*

## 2.2 Active Learning

For our purposes, active learning is the process of learning a sequence of hypotheses  $\mathcal{H}_1 \mathcal{H}_2 \dots \mathcal{H}_F$ , such that their behavior converges to some target automaton (DFA, or Mealy machine). The key components are illustrated in Fig. 4.

<sup>1</sup> Extensions and equivalences may be defined as in [2] (such as implication:  $\implies$ , globally  $G$ , and future:  $F$ ).

*Learner*: an algorithm that can form hypotheses based on queries and counterexamples.

*Equivalence oracle* ( $=$ ): an oracle that decides whether two languages are equal. The oracle decides between the language of the current hypothesis of the learner, and the language of the SUL. If the languages are not equivalent the oracle will provide a counterexample that distinguishes both languages. The language of the SUL is a set of finite traces.

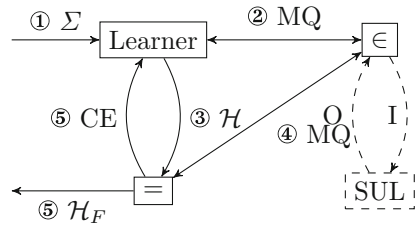


Fig. 4. Active learning

*Membership oracle* ( $\in$ ): an oracle that decides whether or not a word is a member of the language of the SUL.

*SUL*: In the case an active learning algorithm is applied to an actual system, a SUL interface is used that can *step* through a system, to answer membership queries. In the LearnLib, the SUL interface exposes the methods `pre` and `post` that can reset a system (i.e. put it back to the initial state), `step` that stimulates the system with one input symbol and returns the corresponding output, `canFork` and `fork` that may fork a SUL, i.e. provide some copy (that behaves identically to) a system. In active learning, this is used to pose queries in parallel. We will show it is useful for performing state equivalence checks in BBC too.

**Definition 6 (query).** Given a DFA  $\mathcal{D} = \langle S, s_0, \Sigma, \delta, F \rangle$ , and a SUL, a query is a function  $q: \Sigma^* \rightarrow \mathbb{B}$ , where  $\mathbb{B} = \{\perp, \top\}$  denotes the set of Booleans, indicating whether the input word is in the language of the SUL or not.

*Example 4 (Active Learning).* Given an alphabet  $\Sigma = \{a, b\}$ , and a DFA  $\mathcal{D}$  to be learned such that  $L(\mathcal{D}) = (ab)^*a?$ , an active learning algorithm could first produce the hypothesis  $\mathcal{D}_1$  in Fig. 5a (the trap state is explicit), where the language accepted is  $L(\mathcal{D}_1) = a^*$ . At some point the equivalence oracle generates  $aa \in \Sigma^*$ , and performs the membership query  $q(aa) = \perp$ . The equivalence oracle recognizes that  $aa \in L(\mathcal{D}_1)$ , and concludes it found a counterexample to  $\mathcal{D}_1$ . The learner refines  $\mathcal{D}_1$ , and produces the final hypothesis in Fig. 5b. Note that this example hides the complexity of actually refining the hypothesis. In the LearnLib refining a hypothesis is done with the method `Learner.refineHypothesis()` that accepts a query (counterexamples) and subsequently poses additional membership queries. More details on refining hypotheses are outside of the scope of this paper; they can be found in e.g. [1, 27].

Finding a counterexample to the current hypothesis by means of an equivalence oracle is expensive in terms of time. In the worst-case the equivalence oracle has to try out all words of maximum length  $n$  in  $\Sigma^n$ . Some smart equivalence oracles (e.g. ones using the partial W-method [8]) can find a counterexample quite quickly, if there is one. However, the number of membership queries to

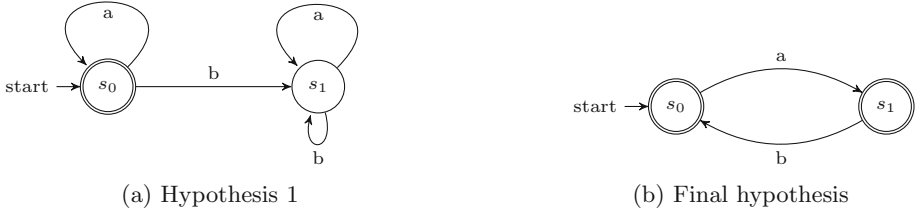


Fig. 5. Active learning

find the counterexample is still orders of magnitudes larger than the size of the hypothesis. E.g. any word of maximum length 2 that could serve as a counterexample for the first hypothesis in Example 4 is in  $\{\epsilon, a, b, aa, ab, ba, bb\}$ . When hypotheses grow larger, the set of possible counterexamples grows with an even larger degree.

### 2.3 Black-Box Checking

Compared to active learning, BBC (Fig. 6) adds a procedure that checks a set of properties  $\{P_1, \dots, P_n\}$  on each hypothesis produced by the Learner. The components added are as follows.

*Model checker* ( $\models$ ): an algorithm that checks whether an hypothesis satisfies a property. If the hypothesis does not satisfy the property it provides some counterexamples to the property. The language of the counterexamples is a subset of the language of the checked hypothesis.

*Emptiness oracle* ( $\emptyset$ ): an oracle that decides whether the intersection of two languages is empty. The oracle decides between the language of the counterexamples given by the model checker, and the language of the SUL. If the intersection is not empty it will provide a counterexample, which is a word in the intersection and as such, a counterexample to the property checked by the model checker.

*Inclusion oracle* ( $\subseteq$ ): an oracle that decides whether one language is included in another. The oracle decides whether the language of the counterexamples given by the model checker is included in the language of the SUL. If the language is not included, the oracle will provide a counterexample such that it is a word not in the language of the SUL, and thus a counterexample to the current hypothesis. One can view the combination of the *model checker*, *emptiness oracle*, and *inclusion oracle* as a *black-box oracle*.

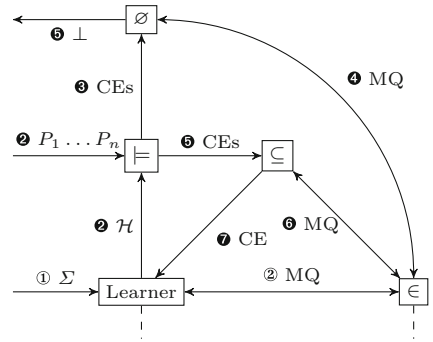


Fig. 6. Black-box checking extension

In traditional active learning there are two kinds of sets of membership queries; *learning queries* (done by the learner) and *equivalence queries* (done by the equivalence oracle). With BBC there are two more types of queries; *inclusion queries* (done by the inclusion oracle), and *emptiness queries* (done by the emptiness oracle). The decision between performing inclusion queries, and emptiness queries depends on whether the property can be falsified with the current hypothesis. We generalize both to *model checking queries*. The key observation why adding properties to verify to the learning algorithm can be useful, follows from the observation that black-box checking queries are very cheap compared to equivalence queries. Given an alphabet  $\Sigma$ , a naive equivalence oracle has to perform arbitrary membership queries for words in  $\Sigma^*$ , while the black-box oracle has to perform only membership queries for a subset of the language of the current hypothesis.

Given that black-box checking queries are much cheaper than equivalence queries a sketch of the black-box checking algorithm (Figs. 5 and 6) is as follows. Initially (①) the learner constructs a hypothesis using membership queries (②). This hypothesis is, together with a set of properties, given to the model checker (②). If the model checker finds counterexamples for a property and the current hypothesis, the counterexamples are given to the emptiness oracle (③). The emptiness oracle performs membership queries (④) to try to find a counterexample from the model checker that is not spurious. If a real counterexample for a property is found, it is reported to the user (⑤), and the property is not considered for future hypotheses. Otherwise, there could be a spurious one, and thus the set of counterexamples are given to the inclusion oracle. The inclusion oracle performs membership queries (⑥) to find a counterexample for the current hypothesis (⑦), the learner performs membership queries (②) to complete the next hypothesis. If the hypothesis is refined, the black-box oracle repeats steps (②, ..., ⑦) until the model checker can not find any new counterexample. In the latter case we enter the traditional active learning loop (Fig. 4): the equivalence oracle tries to find a counterexample for the current hypothesis (③) using membership queries (④). If a counterexample is found (⑤) the learner will construct the next hypothesis using membership queries (②) and the black-box oracle is put back to work. If the equivalence does not find a counterexample (④) the final hypothesis is reported to the user. Note that a black-box oracle can be implemented in two ways. The black-box oracle can first try to find a counterexample for every property before finding a refinement for the current hypothesis. The second implementation finds a counterexample for a single property and if such a counterexample does not exist, find a counterexample for the current hypothesis, before checking the next property. One may favor the first implementation if there is a high chance a property can be disproved with the current hypothesis, or refining the current hypothesis becomes quite expensive.

*Example 5 (Black-Box Checking).* Consider again the first hypothesis  $\mathcal{D}$ , produced by an active learning algorithm from Fig. 5a, that accepts the language  $a^*$ , and the LTL formula  $\phi = \chi(\text{letter} = b)$ , from Example 3. An LTL model checker checks whether  $\mathcal{D} \models \phi$ . The model checker concludes  $\mathcal{D}$  does not model



$\phi$ , and produces the lasso  $a^\omega$  as a counterexample. The model checker unrolls the loop of the lasso an arbitrary number of 3 times, and provides the singleton language  $L(CEs) = \{aaa\}$  to the emptiness oracle ( $\emptyset$ ). The emptiness oracle checks whether the intersection of the language of the SUL ( $L(SUL)$ ), and  $L(CEs)$  is empty. To this end, a membership query  $q(aaa) = \perp$  is performed. This means indeed  $L(SUL) \cap L(CEs) = \emptyset$  and the property can not be falsified. Next,  $L(CEs)$  is given to the inclusion oracle ( $\subseteq$ ) that checks  $L(CEs) \subseteq L(SUL)$ . To this end the inclusion oracle performs the same membership query  $q(aaa) = \perp$ . The inclusion oracle concludes that  $L(CEs) \not\subseteq L(SUL)$ , and thus provides  $aaa \notin L(SUL)$  as a counterexample to the learner. The essence of this example is that Fig. 5a, can be refined without performing any equivalence query. This example (like Example 4) hides to complexity of refining a hypothesis too. Refining a hypothesis in the LearnLib in the context of BBC can also be done with `Learner.refineHypothesis()`.

### 3 Sound Black-Box Checking

The main contribution is (1.) the concept of sound BBC, that involves checking whether a SUL accepts a lasso-shaped infinite word, and (2.) an overview of the implementation in the LearnLib.

#### 3.1 Validating Lassos with State Equivalence

Making the BBC procedure sound involves checking whether infinite lasso-shaped words given as counterexamples by the model checker are accepted by the SUL. Obviously in practice checking whether a SUL accepts an infinite word is impossible. However, this can be resolved if one considers what goes on inside a black-box system. We need to check if the SUL also exhibits a particular lasso through its state space when stimulated with a finite word (that also produces the same output as given by the model checker). This can be achieved by observing particular states the SUL evolves through when stimulated. Note that this view of a SUL is still quite a black-box view; we only record the states, we do not *enforce* the SUL to move to a particular state. We introduce a new notion of a *query*, namely an  $\omega$ -*query*, which in addition to the input word and output of the SUL also contains which states need to be recorded, and which states where actually visited. Compared with traditional BBC, sound BBC requires an emptiness oracle for lassos, denoted  $\emptyset_\omega$ , and a membership oracle for lassos, denoted  $\in_\omega$ .

**Definition 7 ( $\omega$ -query).** *Given a DFA  $\mathcal{D} = \langle S, s_0, \Sigma, \delta, F \rangle$ , and another set of states  $Z$  from the SUL, an  $\omega$ -query is a function  $q_\omega: \Sigma^* \times 2^{\mathbb{N}} \rightarrow \mathbb{B} \times Z^*$ , where  $\mathbb{B} = \{\perp, \top\}$  denotes the set of Booleans, indicating whether the input word is in the language of the SUL or not,  $2^{\mathbb{N}}$  the set of possible symbol indices after which a state has to be recorded, and  $Z^*$  a sequence of possible recorded states. A definition for an  $\omega$ -query for Mealy machines is analogous.*

*Example 6 ( $\omega$ -query).* An example property that does not hold for the final DFA  $\mathcal{D}$  in Fig. 5b is  $\phi = (\text{letter} = b)$ . Whenever a model checker determines whether  $\mathcal{D} \models \phi$ , it may give the lasso  $l = a(ba)^\omega$  as a potential counterexample for  $\phi$ . The language  $L(\text{CEs}) = \{l\}$  is given to the lasso emptiness oracle  $\emptyset_\omega$ , which will unroll the loop of the lasso an arbitrary number of 3 times, and asks the omega membership oracle ( $\in_\omega$ ) for  $q_\omega(\text{abababa}, \{1, 3, 5\}) = (\top, s_1 s_1 s_1)$ . Here it is clear the SUL cycles through state  $s_1$ , and thus accepts the infinite lasso-shaped word  $l$ .

In general, determining whether a state sequence is a closed loop can be done with Definition 8 (we record states at the beginning of each loop iteration). This definition allows us to check whether a SUL accepts a lasso in the most general way. E.g. to check whether a SUL accepts lasso  $p(q_1 q_2 \dots q_n)^\omega$  in a finite number of steps, we also check if the SUL accepts structurally different shaped (but equivalent) lassos, such as  $p q_1 (q_2 \dots q_n q_1)^\omega$ ,  $p(q_1 q_2 \dots q_n q_1 q_2 \dots q_n)^\omega$  etc.

**Definition 8 (closed-loop).** *Given an  $\omega$ -query  $q_\omega(pq^n, I) = \{\top, s\}$ , a state sequence  $s = s_0 s_1 \dots s_n$  is a closed-loop iff  $n > 0$ , and  $\exists 0 \leq i < j \leq n: s_i = s_j$ , and  $I = \{|p|, |p| + |q|, \dots, |p| + |q| \cdot n\}$ .*

### 3.2 Implementation in the LearnLib

We extend the interface of the LearnLib following Fig. 6, with a new type of query, and more oracles. The purpose of queries is to have a well defined way of exchanging information between the learner and the SUL. Oracles find counterexamples to claims, that may in practice, be undecidable to do.

**SUL:** The SUL interface is extended with methods `boolean canRetrieveState()` indicating whether states can actually be observed in the SUL, if this is not possible then sound BBC is not possible, `Object getState()` returning the current state of the SUL, `boolean deepCopies()` indicating whether the object returned by `getState()` is a deep copy.

**ModelChecker:** A `ModelChecker` may find a counterexample to a property and hypothesis. A counterexample is a subset of the language of the hypothesis. `LTSmin` [4, 15] is an available implementation of a `ModelChecker` for LTL in the LearnLib.

**OmegaQuery:** An `OmegaQuery` is a specialization of a `Query`. An answered `Query` contains information about whether a word is in the language of the SUL. An `OmegaQuery` specializes this behavior to infinite words.

**OmegaMembershipOracle:** An oracle that decides whether an infinite word is in the language of the SUL. To this end it poses `OmegaQueries`. There are several implementations available; one that simulates DFAs and Mealy machines, and one that wraps around a SUL.

**EmptinessOracle:** An `EmptinessOracle` generates words that are in a given automaton, and tests whether those words are also in the SUL. The current implementation, generates words in a breadth-first manner. A limit can be placed on the maximum number of words. An `EmptinessOracle` is used to

check whether any word in the language given as a counterexample by the `ModelChecker` is present in the SUL. A specialization of an `EmptinessOracle` is a `LassoEmptinessOracle` that uses `OmegaQueries` to check whether infinite lasso-shaped words are not in the SUL.

**InclusionOracle:** Similar to the `EmptinessOracle`; it generates a limited number of words in a breadth-first manner, but checks whether words are *in* the language of the SUL. Note that both of these oracles may perform the same queries; this is a practical issue and is usually resolved by using a `SULCache` so that in case of a cache-hit the SUL is not stimulated. The `InclusionOracle`, and `EmptinessOracle` may have different strategies (BFS vs. DFS), and hence are not merged together into a single oracle. Separation of concerns (finding a counterexample to the current hypothesis, vs. finding a counterexample to a property), is also considered a good design principle.

**BlackBoxProperty:** a `BlackBoxProperty` is a property for a black-box system. It may be *disproved*, or used to find a *counterexample* to the current hypothesis. To these ends, it requires a `ModelChecker`, `EmptinessOracle`, `InclusionOracle`, and the property itself, such as an LTL formula. Note that LTL counterexamples for safety properties not necessarily exhibit a lasso structure. A future improvement could exploit this and hence the `EmptinessOracle` is given to `BlackBoxProperty`, and not to a `BlackBoxOracle`.

**BlackBoxOracle:** an oracle that disproves a set of `BlackBoxProperties`, or find a counterexample to the current hypothesis in the same set of `BlackBoxProperties`. Currently, there are two implementations available. One implementation iterates over the set of properties that are still unknown, and tries to disprove any of them before refining the current hypothesis. The other implementation iterates over the set of properties that are still unknown, and before disproving a next property it first tries to refine the current hypothesis with the current property. Both implementations at their core compute a least fixed-point of a set of properties they can not disprove. The latter implementation is used in the experiments later. In the case where an `OmegaMembershipOracle` wraps around a SUL there are two implementations available, based on the implementation of `SUL.deepCopies()`. If a SUL does not make a deep copy of the state of the SUL it could be the case that if `SUL.step()` is executed, a previously obtained state with `SUL.getState()` would also be modified, e.g. the assertion in the Java snippet

```
Object o1 = SUL.getState(); int hc = o1.hashCode(); SUL.step();
    assert o1.hashCode()== hc;
```

may not hold. To resolve this; if `SUL.deepCopies()` does not hold, then `SUL.forkable()` must hold. Two instances of a SUL are used, i.e. one regular instance, and a forked instance to compare two states. More specifically an `OmegaMembershipOracle` that wraps around a SUL that does not make deep copies of states in fact uses hash codes of states, and if the hash codes of two states are equal, the `OmegaMembershipOracle` will step one instance of the SUL through the access sequence of one state, and the forked instance of the SUL through the access sequence of the second state.

In case `SUL.deepCopies()` *does* hold, checking equality of two states is straightforward; one can simply invoke `Object.equals()` on the two states. Listing 1.1 shows how the running example can be implemented in the LearnLib. Note that we show how a membership oracle can answer queries by simulating a DFA. In Sect. 5 we show how one can learn a Mealy machine by implementing LearnLib’s SUL interface.

**Listing 1.1.** Black-box checking in the LearnLib

```
// define the alphabet
Alphabet sigma = Alphabets.characters('a', 'b');
// create the running example DFA
DFA dfa = AutomatonBuilders.newDFA(sigma).
    withInitial("q0").withAccepting("q0").withAccepting("q1").
    from("q0").on('a').to("q1").from("q1").on('b').to("q0").create();
// create an omega membership oracle, that simulates the DFA
DFAOmegaMembershipOracle oMO = new DFASimulatorOmegaOracle(dfa);
// create a regular membership oracle
DFAMembershipOracle mO = oMO.getDFAMembershipOracle();
// create an equivalence oracle that uses the partial W-method
DFAEquivalenceOracle eqO = new DFAWpMethodEQOracle(3, mO);
// create a TTT learner
DFALearner learner = new TTTLearnerDFA(sigma, mO, LINEAR_FWD);
// create a parser that translates data between LTSmin and the LearnLib
Function<String, Character> edgeParser = s -> s.charAt(0);
// create an LTSmin model checker
DFAModelCheckerLasso modelChecker = new
    LTSminLTLDFABuilder().withString2Input(edgeParser).create();
// create an emptiness oracle for lassos
DFALassoEmptinessOracle emO = new DFALassoDFAEmptinessOracle(oMO);
// create an inclusion oracle
DFAInclusionOracle inO = new DFABreadthFirstInclusionOracle(1, mO);
// create the black-box property from the running example
DFABlackBoxProperty lt1 = new DFABBPropertyDFALasso(modelChecker, emO,
    inO, "X letter==\`b\`");
// create the black-box oracle with the singleton set of properties
DFABlackBoxOracle bBO = new CExFirstDFABBOracle(lt1);
// create a black-box checking experiment
DFABBCExperiment e = new DFABBCExperiment(learner, eqO, sigma, bBO);
// run the experiment
e.run();
// assert we have the correct result
assert findSeparatingWord(dfa, e.getFinalHypothesis(), sigma) == null;
```

## 4 Related Work

Related work in context of this work can be found in three main areas. First, there is a tool that already does BBC, called `LBTest` [21]. Second, other than the LearnLib there is another active learning framework called `libalf` [5]. Third, aside from `LTSmin` there are other model checkers such as `NuSMV` [6], and `SPIN` [9]. Currently, `LBTest` is not Free and Open Source Software (FOSS). The LearnLib on the other hand is licensed under the Apache 2 license and thus freely available, even for commercial use. This argument is important because BBC is very successful when applied to industrial critical systems [17, 19]. Our new implementation in the LearnLib is also licensed under the Apache 2 license. Our reasoning for implementing BBC in the LearnLib, and not `libalf` is that LearnLib is actively maintained, while `libalf` is not.

We choose to select the LTSmin [15] model checker, because LTSmin, similar to the LearnLib has a liberal BSD license, and is still actively maintained. Compared to NuSMV, LTSmin has an explicit-state model checker, while NuSMV is a symbolic model checker using BDDs. In principle NuSMV would also suffice as a model checker in this work. We have designed our BBC approach in such a way that in the future integrating NuSMV with the LearnLib is easy. Another popular model checker is SPIN. The disadvantage of using the SPIN model checker is that the counterexamples it produces are state-based, while active learning algorithms require action-based counterexamples [26].

BBC is not new to the LearnLib, several years ago a similar study was performed, named *dynamic testing* [24]. Recently new active learning algorithms such as ADT [7], and TTT [13] have been added to the LearnLib, and their performance in the context of BBC is still unknown. Both ADT, and TTT may very well compare to the main learning algorithm Incremental Kripke Learning (IKL) [20] in LBTest, which is a so-called incremental learning algorithm. Incremental learning algorithms try to produce new hypotheses more quickly, in order to reduce the number of learning queries. Traditional active learning algorithms, such as  $L^*$  produce fewer hypotheses, where each new hypothesis requires more learning queries. The latter makes sense in the context of active learning, because this minimizes the number of equivalence queries necessary. In the context of active learning incremental learning algorithms may actually degrade performance; while they may perform well in the number of learning queries, they may require more equivalence queries to refine the hypotheses, resulting in longer run times, see [11, Sect. 5.5]. In BBC model checking queries can be used to refine hypotheses. Model checking queries are negligible compared to equivalence queries [20], making the ADT, and TTT algorithms excellent candidates for a BBC study.

## 5 Results

BBC in the presence of a good amount of LTL formulae can greatly reduce the number of learning queries, and equivalence queries required to disprove the LTL formulae compared to active learning. Note that, although BBC introduces additional model checking queries (performed by the equivalence oracle, or inclusion oracle), these model checking queries are dwarfed by the amount of equivalence queries (and even learning queries). We will thus refrain from reporting the amount model checking queries here (they can be found online<sup>2</sup>, alongside reproduction instructions). What we will show is the following.

- How many learning queries, and equivalence queries it takes to disprove as many LTL formulae as possible in the traditional active learning setting. This means evaluating all LTL formulae after active learning algorithms produce the final hypothesis.

<sup>2</sup> <https://github.com/Meijuh/NFM2018BBC>.

- The amount of learning queries, and equivalence queries in the BBC setting to disprove as many LTL formulae as possible.

Currently there are eight active learning algorithms implemented in the LearnLib for Mealy machines, which are as follows: ADT [7], DHC [22], Discrimination Tree [10],  $L^*$  [1], Kearns and Vazirani [16], Maler and Pnueli [18], Rivest and Schapire [25], and TTT [13]. To investigate the performance of these algorithms in a BBC setting we take problem instances, and LTL formulae from the 2017 RERS challenge. The Rigorous Examination of Reactive Systems (RERS) challenge<sup>3</sup> is a yearly recurring verification challenge [14]. There are two main categories. In one category one has to solve properties for problems which are parallel in nature [29]. The other category involves sequential problems [28]. The RERS sequential problems are provided in Java (among others); the Java problem structure is given in Listing 1.2.

One can see that it is straightforward to actively learn a Mealy machine from a `Problem` instance. The *alphabet* is specified with the field `String[] inputs`. The *state* of a problem instance is determined by the valuations of some instance variables (`a175`, `a52`, `a176`, `a166`, `a167`, and `a62`). An *input* can be given to the `calculateOutput` method, which returns an *output*. The problem instance can be *reset* with the `reset()` method. A SUL implementation of a RERS `Problem` is easy: `SUL.post()` invokes `Problem.reset()`, `SUL.step()` invokes `Problem.calculateOutput()`.

To achieve *sound* BBC, we must be able to retrieve the current state of a `Problem` instance. We choose not to make deep copies of a state of a `Problem`, hence `SUL.deepCopies()` does not hold. This means an `OmegaMembershipOracle`, must use `Object.hashCode()`, and `Object.equals()`. These methods can be easily generated with project Lombok<sup>4</sup>, by annotating a class with `@EqualsAndHashCode`. Lastly, the SUL can be forked by creating a new SUL instance, with a new `Problem` instance.

We benchmark the LearnLib active learning algorithms with nine different RERS problems from the 2017 RERS challenge in a BBC setting. Each problem comes with 100 different LTL formulae, where typically approximately half of the formulae hold, and the other half does not hold. When active learning algorithms are able to learn the complete Mealy machine, this Mealy machine will be minimal. In case of the RERS problems the size of those Mealy machine

**Listing 1.2.** RERS structure

```
@EqualsAndHashCode(exclude =
    {"inputs"})
public class Problem {
    ...
    public String[] inputs =
        {"B", "E", "C", "A", "D"};

    private int a175 = 6;
    private int a52 = 9;
    private int a176 = 7;
    private String a166 = "e";
    private String a167 = "e";
    private String a62 = "f";

    public String
        calculateOutput(String
            i){ }
    public void reset(){ }
    ...
}
```

<sup>3</sup> <http://rers-challenge.org>.

<sup>4</sup> <https://projectlombok.org>.

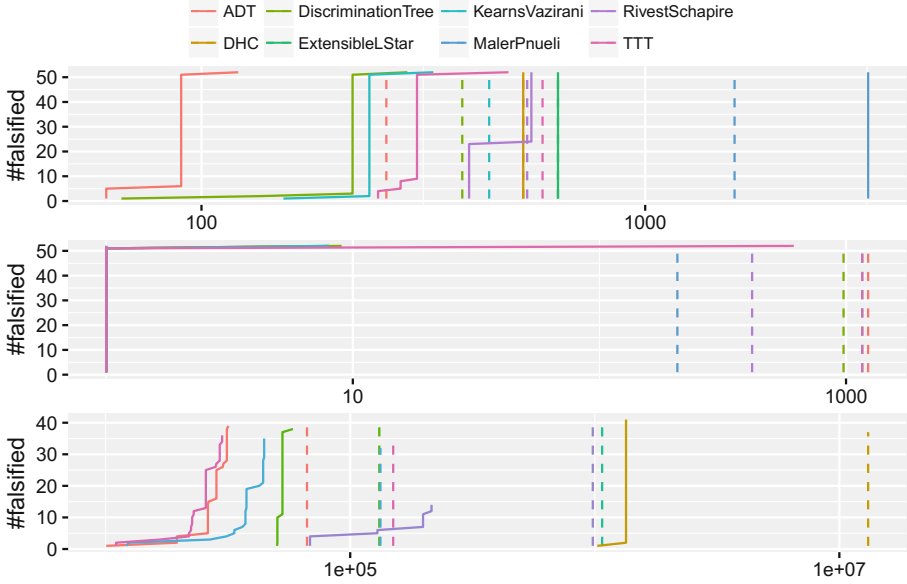


Fig. 7. Experimental results (Color figure online)

range from tens of states to several thousands. Additionally this requires a few hundred to several thousand learning queries, and several thousand to millions equivalence queries. In Fig. 7 the top graph shows the legend. The second graph shows the number of learning queries for the smallest RERS problem, and the third graph the number of equivalence queries. The last graph shows the number of learning queries for the largest RERS problem. The x-axes show on a logarithmic scale the number of queries required to disprove a certain number of properties. The y-axes show the amount of properties that are disproved. A dashed line shows the relation between queries and falsified properties in an active learning setting, while a normal line shows the relation in a BBC setting. The further a line appears to the left; the better the algorithm. A dashed line is always purely vertical, because active learning algorithms do not disprove properties on-the-fly (i.e. the same number of queries is required to disprove all properties). In the case of BBC (uninterrupted lines) properties are disproved on-the-fly. This means fewer queries may be required to disprove the first properties. One can also see that in some cases an uninterrupted line, and dashed line of the same color are not equally high. This means that within the used timeout of 1 h active learning did not construct the complete hypothesis, and thus disproves fewer properties. Interestingly, almost all algorithms use fewer learning queries when used in the context of BBC. And even more interesting, some algorithms only use equivalence queries to disprove the last few properties. Obviously this is a great result. Figure 7 also shows that (as suspected) the incremental TTT,

and ADT algorithms produce more equivalence queries compared to a classic algorithm like Rivest and Schapire. The performance of the eight algorithms is quite consistent throughout the larger problem instances. The ADT algorithm seems to perform really well, but the TTT is quite competitive too, this can be seen especially in the largest RERS problem. Also the last graph<sup>5</sup> shows that TTT seems to need fewer learning queries, but ADT seems to be able to disprove more properties within 1 h. The great performance of ADT is particularly interesting since it is only developed recently. The ADT algorithm is developed to reduce the number of resets of the SUL. Now it seems to be the best choice for BBC too among the benchmarked algorithms and RERS problem instances.

## 6 Conclusion

We have presented a black-box checking implementation for the LearnLib. This includes a novel sound approach for liveness LTL properties, where we can check if a system-under-learning accepts an infinite lasso-shaped word. This contrasts the original proposal where an (hard to guess) upper-bound on the number of states of the system-under-learning is assumed. Our implementation is available under a liberal free and open source license, such that it can be put to practice quite easily. Our results (Fig. 7) show that recently added ADT, and TTT active learning algorithms perform the best in a black-box checking setting. In contrast to some other learning algorithms in the LearnLib, ADT, and TTT are incremental learning algorithms, meaning they construct more hypotheses while using less learning queries. In an active learning setting this may degrade performance, because more equivalence queries are required. In a black-box checking setting this appeared to be an advantage, because model checking queries replace expensive equivalence queries. Further work may show how ADT, and TTT compare with the IKL algorithm in LBTest. Software testers now have a free ease-of-use *sound* black-box checking implementation available for industrial use cases. Future work may show whether additional model checkers such as NuSMV provide comparable results, or if there exist different valuable strategies for finding (spurious) counterexamples to properties. In our case study we applied a perfect state equivalence function to the RERS problems, it would be interesting to apply our approach to cases where only part of the state can be observed, or when the SUL is hardware, instead of software.

**Acknowledgements.** We want to thank the developers of the AutomataLib, and the LearnLib; without the extraordinary design of those tools, this work would not have been possible. Furthermore, we would like to thank Frits Vaandrager for his useful feedback on a draft version of this paper.

---

<sup>5</sup> Maler and Pnueli is not shown, because it was not able to disprove a single property.





## References

1. Angluin, D.: Learning regular sets from queries and counterexamples. *Inf. Comput.* **75**(2), 87–106 (1987)
2. Baier, C., Katoen, J.: *Principles of Model Checking*. MIT Press, Cambridge (2008)
3. Belinfante, A.: *JTorX: exploring model-based testing*. Ph.D. thesis, University of Twente, Enschede, Netherlands (2014)
4. Bloemen, V., van de Pol, J.: Multi-core SCC-based LTL model checking. In: Bloem, R., Arbel, E. (eds.) *HVC 2016*. LNCS, vol. 10028, pp. 18–33. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-49052-6\\_2](https://doi.org/10.1007/978-3-319-49052-6_2)
5. Bollig, B., Katoen, J.-P., Kern, C., Leucker, M., Neider, D., Piegdon, D.R.: libalf: the automata learning framework. In: Touili, T., Cook, B., Jackson, P. (eds.) *CAV 2010*. LNCS, vol. 6174, pp. 360–364. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-14295-6\\_32](https://doi.org/10.1007/978-3-642-14295-6_32)
6. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) *CAV 2002*. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). [https://doi.org/10.1007/3-540-45657-0\\_29](https://doi.org/10.1007/3-540-45657-0_29)
7. Frohme, M.: *Active automata learning with adaptive distinguishing sequences*. Master’s thesis, Technische Universität Dortmund (2015)
8. Fujiwara, S., von Bochmann, G., Khendek, F., et al.: Test selection based on finite state models. *IEEE Trans. Softw. Eng.* **17**(6), 591–603 (1991)
9. Holzmann, G.J.: *The SPIN Model Checker - Primer and Reference Manual*. Addison-Wesley, Boston (2004)
10. Howar, F.: *Active learning of interface programs*. Ph.D. thesis, Dortmund University of Technology (2012)
11. Isberner, M.: *Foundations of active automata learning: an algorithmic perspective*. Ph.D. thesis, Technical University Dortmund, Germany (2015)
12. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib - a framework for active automata learning. In: Kroening, D., Păsăreanu, C.S. (eds.) *CAV 2015*. LNCS, vol. 9206, pp. 487–495. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-21690-4\\_32](https://doi.org/10.1007/978-3-319-21690-4_32)
13. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: a redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) *RV 2014*. LNCS, vol. 8734, pp. 307–322. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-11164-3\\_26](https://doi.org/10.1007/978-3-319-11164-3_26)
14. Jasper, M., Fecke, M., Steffen, B., et al.: The RERS 2017 challenge and workshop (invited paper). In: *SPIN, Santa Barbara, CA, USA, 10–14 July 2017*, pp. 11–20 (2017)
15. 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](https://doi.org/10.1007/978-3-662-46681-0_61)
16. Kearns, M.J., Vazirani, U.V.: *An Introduction to Computational Learning Theory*. MIT Press, Cambridge (1994)
17. Khosrowjerdi, H., Meinke, K., Rasmusson, A.: Learning-based testing for safety critical automotive applications. In: Bozzano, M., Papadopoulos, Y. (eds.) *IMBSA 2017*. LNCS, vol. 10437, pp. 197–211. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-64119-5\\_13](https://doi.org/10.1007/978-3-319-64119-5_13)

18. Maler, O., Pnueli, A.: On the learnability of infinitary regular sets. *Inf. Comput.* **118**(2), 316–326 (1995)
19. Meinke, K.: Learning-based testing of cyber-physical systems-of-systems: a platooning study. In: Reinecke, P., Di Marco, A. (eds.) *EPEW 2017*. LNCS, vol. 10497, pp. 135–151. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-66583-2\\_9](https://doi.org/10.1007/978-3-319-66583-2_9)
20. Meinke, K., Sindhu, M.A.: Incremental learning-based testing for reactive systems. In: Gogolla, M., Wolff, B. (eds.) *TAP 2011*. LNCS, vol. 6706, pp. 134–151. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-21768-5\\_11](https://doi.org/10.1007/978-3-642-21768-5_11)
21. Meinke, K., Sindhu, M.A.: LBTest: a learning-based testing tool for reactive systems. In: *ICST, Luxembourg, 18–22 March 2013*, pp. 447–454 (2013)
22. Merten, M., Howar, F., Steffen, B., Margaria, T.: Automata learning with on-the-fly direct hypothesis construction. In: Hähnle, R., Knoop, J., Margaria, T., Schreiner, D., Steffen, B. (eds.) *ISoLA 2011*. CCIS, pp. 248–260. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-34781-8\\_19](https://doi.org/10.1007/978-3-642-34781-8_19)
23. Peled, D.A., Vardi, M.Y., Yannakakis, M.: Black box checking. *J. Autom. Lang. Comb.* **7**(2), 225–246 (2002)
24. Raffelt, H., Steffen, B., Margaria, T.: Dynamic Testing Via Automata Learning. In: Yorav, K. (ed.) *HVC 2007*. LNCS, vol. 4899, pp. 136–152. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-77966-7\\_13](https://doi.org/10.1007/978-3-540-77966-7_13)
25. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. *Inf. Comput.* **103**(2), 299–347 (1993)
26. Sindhu, M.A.: Algorithms and tools for learning-based testing of reactive systems. Ph.D. thesis (2013)
27. Steffen, B., Howar, F., Merten, M.: Introduction to active automata learning from a practical perspective. In: Bernardo, M., Issarny, V. (eds.) *SFM 2011*. LNCS, vol. 6659, pp. 256–296. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-21455-4\\_8](https://doi.org/10.1007/978-3-642-21455-4_8)
28. Steffen, B., Isberner, M., Naujokat, S., et al.: Property-driven benchmark generation: synthesizing programs of realistic structure. *STTT* **16**(5), 465–479 (2014)
29. Steffen, B., Jasper, M., et al.: Property-preserving generation of tailored benchmark Petri nets. In: *ACSD, Zaragoza, Spain, 25–30 June 2017*, pp. 1–8 (2017)
30. Timmer, M., Brinksma, E., Stoelinga, M.: Model-based testing. In: *Software and Systems Safety - Specification and Verification*, pp. 1–32 (2011)



# Model-Checking Task Parallel Programs for Data-Race

Radha Nakade<sup>1</sup>, Eric Mercer<sup>1</sup>(✉) , Peter Aldous<sup>1</sup> , and Jay McCarthy<sup>2</sup>

<sup>1</sup> Brigham Young University, Provo, UT, USA  
radha.nakade@gmail.com, {egm,aldous}@cs.byu.edu

<sup>2</sup> University of Massachusetts Lowell, Lowell, USA  
jay.mccarthy@gmail.com

**Abstract.** Data-race detection is the problem of determining if a concurrent program has a data-race in some execution and input; it has been long studied and often solved. The research in this paper reprises the problem in the context of task parallel programs with the intent to prove, via model checking, the absence of data-race on any feasible schedule for a given input. Many of the correctness properties afforded by task parallel programming models such as OpenMP, Cilk, X10, Chapel, Habanero, etc. rely on data-race freedom. Model checking for data-race, presented here, is in contrast to recent work using run-time monitoring, log analysis, or static analysis which are complete or sound but never both. The model checking algorithm builds a happens-before relation from the program execution and uses that relation to detect data-race similar to many solutions that reason over a single observed execution. Unlike those solutions, model checking generates additional program schedules sufficient to prove data-race freedom over all schedules on the given input. The approach is evaluated in a Java implementation of Habanero using the JavaPathfinder model checker. The results, when compared to existing data-race detectors in Java Pathfinder, show a significant reduction in the time required for proving data race freedom.

## 1 Introduction

A *data-race* is where two concurrent executions access the same memory location with at least one of the two accesses being a write. It introduces non-determinism into the program execution as the behavior may depend on the order in which the concurrent executions access memory. Data-race is problematic because it is not possible to directly control or observe the run-time internals to know if a data-race exists let alone enumerate program behaviors when one does.

The *data-race detection* problem, given a program with its input, is to determine if there exists an execution containing a data-race. The research presented in this paper is concerned with proving data-race freedom for *task parallel models* that impose structure on parallelism by constraining how threads are created and joined, and by constraining how shared memory is accessed (e.g., OpenMP, Cilk, X10, Chapel, Habanero, etc.). These models rely on run-time environments to implement task abstractions to represent concurrent executions [1–4].

The language restrictions on parallelism and shared memory interactions enable properties like *determinism* (i.e., the computation is independent of the execution) or the ability to *serialize* (i.e., removing all task related keywords yields a serial solution). Such properties only hold in the absence of data-race, which is not always the case since programmers, both intentionally and unintentionally, move outside the programming model.

Data-race detection in task parallel models generally prioritizes performance and the ability to scale to many tasks over a proof of absence. The predominant *SP-bags* algorithm, with its variants, is a dynamic approach that exploits assumptions on task creation and joining for efficient on-the-fly detection with low overhead [5–9]; millions of tasks are feasible with varying degrees of slow-down (i.e., slow-down increases as parallelism constraints are relaxed) [10]. Other approaches use access histories [11, 12] or programmer annotations [13]. Performance is a priority requiring careful integration into complex run-time environments, and solutions are only *complete*, meaning that nothing can be concluded about other executions of the program on the same input.

The research presented in this paper reprises the data-race problem in task parallel models with the intent to prove, via model checking, data race freedom on a given input over all feasible executions. Prior model checking based solutions enumerate schedules that interleave conflicting accesses, meaning at least one access is a write, to shared variables [14–16]. The approach here rather uses techniques from dynamic approaches to build a happens-before relation from a single observed program execution sufficient to prove data-race freedom in all executions that order mutually exclusive regions in the same way as the observed execution [17–19]. As such, the happens-before relation captures in one many of the schedules exhaustively enumerated by the prior model checking solutions. Unlike the dynamic approaches, though, the approach here then generates other program executions necessary to prove data-race freedom over all executions on the input. As a result, in the absence of mutual exclusion, a single program execution is sufficient to prove data-race freedom. In the presence of mutual exclusion, the model checker generates and checks all feasible orderings of the mutually exclusive regions to complete the proof. Underlying this contribution is the fact that we assume the program under test terminates; if such is not the case, then the research in this paper does not directly apply.

The research presented in this paper includes an empirical study of the proposed model checking algorithm for a Java implementation of Habenero with the Java Pathfinder model checker (JPF). Unlike prior solutions, this implementation uses an idealized verification run-time for Habanero rather than a production run-time, does not require internal modifications to JPF, and gives results about the input program that generalize to any language run-time implementation [14–16]. Results over several published benchmarks comparing to JPF’s default race detection using partial order reduction and a task parallel approach with permission regions show the approach here to be more efficient in JPF terms with its inherent overhead. Of course, as with any model checking approach, the intent is to not scale to millions of parallel tasks with hundreds of

```

proc m (var x : int)
  g := 0;
  post r ← p 0 λv. skip;
  [ isolated g := 1 ]
  await r
  return x

proc p (var x : int)
  [ isolated skip; ]
  g := 2;
  return 0

```

(a)

```

public class Example1{
  static int g = 0;
  public static void main(String[] args) {
    g := 0
    finish {
      async { p(0); }
      isolated{ g := 1; }
    }
    public static void p(int x) {
      isolated{ /* skip */ };
      g := 2;
      return 0;
    }
  }
}

```

(b)

**Fig. 1.** A program with data-race. (a) Task parallel. (b) Habanero Java.

mutually exclusive regions; rather, this research assumes that it is possible to provide input to any given program that results in hundreds of tasks and tens of mutually exclusive regions. It further assumes that a data-race freedom proof on the small instance generalizes to the large instance. The primary contributions are thus

- a simple approach to data-race detection in programs that terminate based on constructing a happens-before relation from an execution of a task parallel program;
- a proof that scheduling to interleave mutually exclusive regions is sufficient to prove data-race freedom; and
- an implementation of the approach for Java Habanero in JPF with results from benchmarks comparing to other solutions in JPF.

The rest of this paper is organized as follows: Sect. 2 illustrates the approach in a small example; Sect. 3 defines the computation graph, task parallel programs, and how to build a computation graph from a program execution; Sect. 4 gives a correctness proof; Sect. 5 is the empirical study with a summary of the implementation; and Sect. 7 is the conclusion with future work.

## 2 Example

The approach to data-race detection in this paper is presented in a very simple example. Consider the task-parallel program in Fig. 1(a). The language used is defined in this paper with a formal semantics to facilitate proofs but has a direct expression in most task parallel languages. For example, Fig. 1(b) is the equivalent program in the Habanero Java language.

For Fig. 1, execution begins with the procedure `m`. The variable `g` is global. The `post`-statement creates a new asynchronous task running procedure `p` passing 0 for its parameter. The task handle is stored in the region `r`, also global, and when that task completes and joins with its parent `m`, it runs the  $\lambda$ -expression as a return value handler. In this case, that handler is the no-op `skip`.

The `isolated`-statement runs the statement in its scope in mutual exclusion to other `isolated`-statements. The `await`-statement joins all tasks in region `r` with the task that issued the `await`. The issuer may join with a task in the region if that task is at its `return`-statement. The expression in the `return`-statement is evaluated at the join and the value is passed to the return value handler in the parent context. The parent blocks at the `await`-statement until it has joined with all tasks in the indicated region.

The program in Fig. 1 has a schedule dependent data race. If the scheduler runs the `isolated`-statement in procedure `p` before the `isolated`-statement in procedure `m` then there is a write-write data-race; otherwise, there is no data-race.

Related work in model checking task parallel languages enumerates schedules to interleave the mutual exclusion and to interleave any unprotected shared memory access leading quickly to state explosion [14–16]. These approaches use the happens-before relation to detect data-race but not to reduce the number of considered schedules—every schedule is checked.

The approach in this paper exploits so-called partial order analyses to reduce the number of schedules that must be checked to prove data-race freedom. The approach uses the simple happens-before partial order, [17], but is easily extended to something like weak causally-precedes to further reduce checked schedules [18, 19]. Unlike other partial-order approaches though, a sufficient set of schedules is checked to prove data-race freedom on the given input.

The approach dynamically detects shared memory accesses and uses the language semantics to capture, during execution, the happens-before relation in the form of a computation graph. The left part of Fig. 2 shows the computation graph for the data-race free schedule of the example program. Every node represents a block of sequential operations and edges order the nodes. The thick `p`-labeled line is the result of the `post`-statement creating a new task, and the dashed boxes are the `isolated`-statements. Intuitively, the computation graph is a Hasse diagram with inverted edges—things at the bottom happen-after things at the top—and with extra information on each node to indicate read and write memory locations. Such a graph can be readily checked for data-race in linear time [17].

To reason over all schedules, the approach in this paper first assumes two restrictions common in most task parallel languages: if a return value handler side-effects, then it exists in a region by itself throughout its lifetime, and all tasks are joined at termination in a deterministic order by a implicit enclosing parent task. Under these restrictions, the model checker, to prove data-race freedom, must generate a set of schedules that contains all ways allowed by the

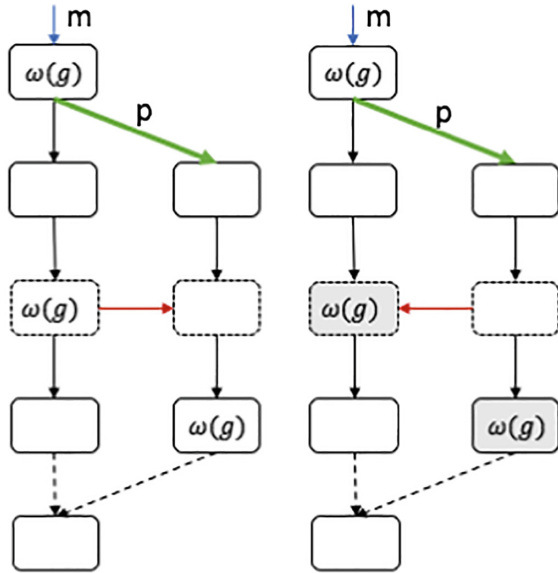


Fig. 2. Two computation graphs for Fig. 1: no data-race on left and data-race right.

program semantics to interleave **isolated**-statements. This result is the main contribution.

The right part of Fig. 2 shows the computation graph for the data-race schedule of the simple example program. Although the two schedules in Fig. 2 are the only schedules that need to be considered by the model checker in this example, the number of interesting schedules grows exponentially in the number of concurrent dependent **isolated**-statements. The growth limits the model checking approach in this paper to programs that can be instantiated on small problem instances; however, in general, a proof on a small problem instance typically generalizes to large problem instances.

### 3 Task Parallel Programs

A task parallel program uses a general programming model to define parallelism. That model, under certain restrictions, captures the semantics of many common task parallel models such as Habanero, X10, Cilk, OpenMP, Chapel, etc. This section first defines the structure of the computation graph, and it then discusses task parallel programs and how a computation graph can be created from an observed execution of a task parallel program. The full semantics for the programming model, and how the computation graph is formally constructed from an execution of the program, is not presented here but can be found at <https://jpf.byu.edu/jpf-hj/>.

### 3.1 Computation Graphs

A Computation Graph for a task parallel program is a directed acyclic graph representing the concurrent structure of the program execution [20]. It is modified here to track memory locations accessed by tasks.

**Definition 1.** A computation graph is a directed acyclic graph (DAG),  $G = (N, E, \rho, \omega)$ , where  $N$  is a finite set of nodes,  $E \subseteq N \times N$  is a set of directed edges,  $\rho : (N \mapsto \mathcal{P}(\mathbf{Globals}))$  maps  $N$  to the unique identifiers for the shared locations read by the tasks,  $\omega : (N \mapsto \mathcal{P}(\mathbf{Globals}))$  maps  $N$  to the unique identifiers for the shared locations written by the tasks, and  $\mathbf{Globals}$  is the set of the unique identifiers for the shared locations.

The graph captures the happens-before relation between nodes:  $< \subset N \times N$ . There is a data race in the graph if and only if there are two nodes,  $n_i$  and  $n_j$ , such that the nodes are concurrent, (i.e.,  $n_i \not\prec n_j \wedge n_j \not\prec n_i$  or, equivalently,  $n_i \parallel_{<} n_j$ ), and the two nodes conflict:

$$\begin{aligned} \text{conflict}(n_i, n_j) = & \rho(n_i) \cap \omega(n_j) \neq \emptyset \vee \\ & \rho(n_j) \cap \omega(n_i) \neq \emptyset \vee \\ & \omega(n_i) \cap \omega(n_j) \neq \emptyset \end{aligned} \quad (1)$$

The condition is readily checked in linear time as mentioned in the previous section.

### 3.2 Programming Model

The programming model is derived from Bouajjani and Emmi for isolated parallel tasks [21]; this variant removes the isolation between tasks with the introduction of shared memory. It additionally restricts task passing to only allow tasks to be passed when a child completes, and those tasks are only passed to the parent. Finally, procedures with side-effecting return value handlers must be the only members of their respective regions.

The surface syntax for the language is given in Fig. 3. A program  $\mathbf{P}$  is a sequence of procedures. Each procedure has a single parameter  $l$  of type  $L$ . Procedures may also reference shared variables taken from a finite set of names which include a special reserved variable `isolate` that is only used by the semantics for mutual exclusion. The body of a procedure is inductively defined by  $\mathbf{s}$ . The expression language,  $e$ , is also abstracted.

The `post`, `await`, `ewait`, and `isolated` statements relate to concurrency and affect the shape of the computation graph; the rest of the statements have their usual sequential meaning. The semantics produce a computation graph as a by-product of reducing the program via rewrites. Two additional data are associated with the computation graph and are used by the semantics in the construction: *last* is a special node used to assert the observed order of `isolated`-statements and  $R : \mathbf{Regs} \mapsto \mathcal{P}(N)$  is a function used to join tasks in a region at synchronization. In general, a function notation is adopted to access members



$$\begin{aligned}
 \mathbf{P} &::= (\text{proc } p \text{ (var } l : L) s) * \\
 \mathbf{s} &::= s; s \mid l := e \mid \mathbf{skip} \mid [\text{if } e \text{ then } s \text{ else } s] \\
 &\mid [\text{while } e \text{ do } s] \mid \text{call } l := p \ e \mid \mathbf{return } e \\
 &\mid \mathbf{post } r \leftarrow p \ e \ d \mid \mathbf{await } r \mid \mathbf{ewait } r \\
 &\mid [\text{isolated } s]
 \end{aligned}$$

**Fig. 3.** The surface syntax for task parallel programs.

of tuples. For example, the members of the  $G = (N, E, \rho, \omega, last, R)$  are accessed as  $N(G)$ ,  $E(G)$ ,  $\rho(G)$ , etc.

The **post**-statement adds a task into a region  $r$ , taken from a finite set of region identifiers, by indicating the procedure  $p$  for the task with an expression for the local variable value  $e$ , and a return value handler  $d$  to run in the context of the parent task. The POST rewrite rule adds two fresh nodes  $n'_0$  and  $n_1$  to the computation graph: node  $n'_0$  represents the statements following **post** and  $n_1$  represents the statements to be executed by the new task:  $N(G') = N(G) \cup \{n'_0, n_1\}$  and  $E(G') = E(G) \cup \{(n_0, n'_0), (n_0, n_1)\}$ . The rule orders both after the current node for the parent,  $n_0$ , in the computation graph. The read set  $\rho$  of node  $n_0$  is updated to include any global variables referenced in the expression,  $\eta(e)$ , for the local parameter value in the new task:  $\rho(G') = \rho(G)[n_0 \overset{\cup}{\mapsto} \eta(e, \sigma)]$ ; meaning that the  $\rho$  function is as before only now it additionally includes the variables read by  $\eta(e, \sigma)$  in the read set for  $n_0$ — $\sigma$  in the store value lookup.

The **await** and **ewait** statements synchronize a task with the sub-ordinate tasks in the indicated region. Intuitively, when a task calls **await** on region  $r$ , it is blocked until all the tasks it owns in  $r$  finish execution. Similarly, when a task issues an **ewait** with region  $r$ , it is blocked until one task it owns in  $r$  completes. A task is termed *completed* when its statement is a **return**-statement.

The **await** rule blocks the execution of the currently executing task until a task in the indicated region completes. A new node to join the two tasks is not created in the computation graph, nor are the two tasks ordered in the sense of join because the choice of task, say its  $t_2$ , in the region is non-deterministic; as such, the computation graph allows tasks in the region to join in any order contrary to the observed reduction by the rule. The rule saves the current node in the graph for  $t_2$ ,  $n(t_2)$ , to join later once the region is empty,  $R(G') = R(G)[r \overset{\cup}{\mapsto} n(t_2)]$ , and it updates the read set for  $t_2$  on the expression in the **return**-statement:  $\rho(G)[n(t_2) \overset{\cup}{\mapsto} \eta(e, \sigma)]$ . The new state adds an **await**-statement after the return value handler statement since the region is not yet empty, and the region valuation function in the new state includes any tasks owned by  $t_2$  (e.g., the new statement context replaces  $S[\mathbf{await } r]$  with  $S[s; \mathbf{await } r]$  where  $s$  in the return value handler).

The AWAIT-DONE rule activates when the last task, let us call it  $t_2$ , in the region is joined. It differs from the AWAIT rule in that it constrains all tasks that have joined in the region to happen-before the new node for the parent in the computation graph, and it does not insert another **await**-statement in the new

state since the region is now empty:  $n' = \text{fresh}()$ ,  $N(G') = N(G) \cup \{n'\}$ , and  $E(G') = E(G) \cup \{(n, n'), (n(t_2), n')\} \cup \{(n_i, n') \mid n_i \in R(G)(r)\}$ .

The **EWAIT** and **EWAIT-DONE** rules follow **AWAIT** and **AWAIT-DONE** respectively only without the recursive statement when the region is not empty since it only needs to wait on a single task to complete. The rules delay the ordering of tasks joined in the region to when the region becomes empty (i.e., the last task joins) just as done for **await**-statements.

The **isolated**-statement provides mutual exclusion relative to other **isolated**-statements. If  $s$  is isolated, then it runs mutually exclusive to any other statements  $s'$  that are also isolated; however,  $s$  does not run mutually exclusive to other non-isolated statements that may be concurrent with  $s$ .

If no other isolated statements are running, then the **ISOLATED** rule updates the **isolated** shared variable to block other tasks from isolating and inserts after the isolated statement  $s$  the new **isolated-end** keyword to reset the shared variable at the end of isolation. The computation graph gets a new node to track accesses in the isolated statement with an appropriate edge from the previous node. A sequencing edge from  $last$  is also added so the previous isolated statement happens before this new isolated statement:  $n' = \text{fresh}()$ ,  $N(G') = N(G) \cup \{n'\}$ , and  $E(G') = E(G) \cup \{(n, n'), (last(G), n')\}$ . As a note,  $last$  is initialized to the initial node when execution starts.

The **ISOLATED-END** rule creates a new node in the computation graph to denote the end of isolation, updates the **isolated** shared variable, and it updates  $last$  to properly sequence any future isolation. **isolated**-statements are totally ordered in the computation graph:  $n' = \text{fresh}()$ ,  $last(G') = n$ ,  $N(G') = N(G) \cup \{n'\}$ , and  $E(G') = E(G) \cup \{(n, n')\}$ .

## 4 Proof of Correctness

For a given program and input, the computation graphs produced by the tree semantics summarized in Sect. 3.1 demonstrate a data race if and only if a data race is possible for the given program and input. Before proving this claim, we formally define data race. The definition of data race depends on definitions for concurrency and conflict.

**Definition 2 (Conflict).** *Two statements conflict if they both access the same shared variable and at least one of them writes to that variable.  $\rho(s)$  and  $\omega(s)$  behave as expected.*

$$\begin{aligned} \text{conflict}(s_i, s_j) = & \rho(s_i) \cap \omega(s_j) \neq \emptyset \vee \\ & \rho(s_j) \cap \omega(s_i) \neq \emptyset \vee \\ & \omega(s_i) \cap \omega(s_j) \neq \emptyset \end{aligned} \quad (2)$$

A state's set of active statements is used to define concurrency.

**Definition 3 (Active statements).** *A state  $\varsigma$  has a set of active statements  $a(\varsigma)$  that corresponds to the next statement to be reduced in each of the active tasks in the state.*

Without loss of generality, we call the program's initial state  $\varsigma_0$ .

**Definition 4 (Concurrency).** *Two statements are concurrent if and only if an execution of the program can result in a state  $\varsigma$  such that both statements are active at the same time:*

$$s \parallel s' \iff \exists \varsigma : \varsigma_0 \xrightarrow{*} \varsigma \wedge \{s, s'\} \subseteq a(\varsigma). \quad (3)$$

A state  $\varsigma$  that satisfies this condition for  $s$  and  $s'$  is called a *witness state* for  $s \parallel s'$ .  $\xrightarrow{*}$  is the transitive closure of the transition relation  $\rightarrow$  defined in the semantics.

**Definition 5 (Data race).** *There is a data race on two statements  $s$  and  $s'$  if and only if they conflict and are concurrent:*

$$DR(s, s') = s \parallel s' \wedge \text{conflict}(s, s'). \quad (4)$$

Two statements that occur in the same thread of execution cannot be concurrent, as exactly one statement is active in each active thread at any point in time. The semantics ensure that no two statements inside of **isolated**-statements can be concurrent, as only one thread may enter an **isolated**-statement at a time.

Before proving the correctness of data races in the computation graph, we observe that only nodes that end in a **post**-statement and nodes for **isolated**-statements can have multiple outgoing edges. In both cases, the nodes reached by these edges are all in distinct threads of execution. Similarly, only **isolated**-statements and nodes following an **await**- or **ewait**-statement (in the parent thread) and following **return**-statements (in child threads) have multiple incoming edges. The edges that converge on these nodes come from distinct threads.

**Lemma 1.** *If two nodes  $n$  and  $n'$  are unordered in a state's computation graph  $G$ , every  $s \in n$  and  $s' \in n'$  are concurrent:*

$$\forall s \in n, s' \in n' : n \parallel_{\prec} n' \implies s \parallel s'. \quad (5)$$

*Proof.* The two nodes  $n$  and  $n'$  are both reachable; otherwise, they would not have been generated in  $G$ . Within a node, it is possible to advance or wait independent of other nodes' behaviors. Accordingly, it is possible to begin at  $\varsigma_0$  and advance until  $s$  is active. Similarly, it is possible to advance until  $s'$  is active. What remains to be proven is whether or not it is possible to reach a state where both  $s$  and  $s'$  are active; in other words, if it is possible for some schedule to reach  $s$  in one task and then to reach  $s'$  in some other task without advancing the first task any further.

By the construction of  $G$ ,  $n$  and  $n'$  must have some least common ancestor  $n_A$  that is also reachable.  $n_A$  must either end in a **post**-statement or be an **isolated**-statement, as the reduction rules only allow these two statements to have multiple outgoing edges. In both cases, the child nodes of  $n_A$  must be in different tasks. Without loss of generality, we say that  $t$  either contains  $n$  or is

some ancestor of the task that does. Similarly, we say that  $t'$  either contains  $n'$  or is an ancestor of the task that does.

We first advance to  $n_A$  on some schedule that does not contradict  $\prec$ . This is possible because  $n_A$ ,  $n$ , and  $n'$  were all generated. Execution may block, necessitating the advancement of other threads; however, no relationship can exist between the thread that leads to  $n'$ , as this would contradict the definition of  $n_A$  as least common ancestor. We advance  $t$  and any relevant children or unrelated threads until reaching  $n$  and then proceed until reaching  $s$ . Then, we advance  $t'$  and any relevant children or unrelated threads until reaching  $n'$  and then proceed until reaching  $s'$ .

Both  $s$  and  $s'$  are active, so they are concurrent.

**Lemma 2 (Soundness of *conflict* over nodes).** *If two nodes conflict, there exists a pair of statements, one from each node, that conflicts:*

$$\text{conflict}(n, n') \implies \exists s \in n, s' \in n' : \text{conflict}(s, s'). \quad (6)$$

*Proof.* If two nodes conflict, it is because  $\rho$  and  $\omega$  were updated in some reduction. More specifically, if  $\rho(n) \neq \emptyset$ , at least one statement  $s \in n$  must read a global variable; the reduction of statements that read a global variable are the only way that  $\rho$  updates. The same reasoning applies to  $\omega$ .

**Lemma 3 (Completeness of *conflict* over nodes).** *If two statements conflict, their respective nodes will conflict.*

$$\forall s \in n, s' \in n' : \text{conflict}(s, s') \implies \text{conflict}(n, n'). \quad (7)$$

*Proof.* If  $s \in n$ , then  $s$  must have been reduced in  $n$ . Per the semantics,  $\rho(n)$  and  $\omega(n)$  must be updated to include reads and writes from  $s$ . Accordingly, nodes conflict whenever statements they include conflict.

**Theorem 1 (Soundness of computation graph over data races).** *If a computation graph reports a data race, there is a data race in the program on the given input.*

*Proof.* By Lemmas 1 and 2.

**Theorem 2 (Completeness of computation graph over data races).** *If there is a data race in the program on the given input, a computation graph generated by the model checker reports a data race.*

*Proof.* We choose, without loss of generality, the first data race manifest along some schedule in the program. Our algorithm reports the first data race it finds and exits. This is consistent with the theorem definition.

The definition of data race states that the two statements must be concurrent, which implies that it is possible to generate a witness state for the two statements' concurrency. As a result, any two statements that conflict and are both reachable will be members of nodes in some computation graph. Because the model checker generates all possible computation graphs, it generates the computation graph created by the witness state. By Lemmas 1 and 3, this computation graph demonstrates the data race.

## 5 Implementation and Results

The model checking approach to data race detection described in this paper has been implemented for Habanero Java (HJ). The implementation uses the verification run-time specifically designed to test HJ programs and play nicely with JPF [16]. The implementation is a set of JPF listeners to create the computation graph and only schedule on **isolated**-statements. It is worth noting that this implementation does not use vector clocks for data-race detection on the generated computation graph but rather uses a more direct, albeit naive, quadratic check via transitive closure.

As a sketch of the implementation, JPF’s VM listeners are used to track various program events related to parallelism. The methods `objectCreated` and `objectReleased` are used to create nodes in the computation graph. The `objectCreated` method is used to track the creation of a new **async** task. The `objectCreated` method detects when a **post** statement executes and adds appropriate edges to the computation graph. Similarly, the `objectReleased` method is used to track when **finish** blocks complete execution. The **await** statement is used to create a node in the graph where the tasks belonging to the **finish** block join. The `executeInstruction` method is used to track memory locations that are accessed by various tasks by updating the node with the location accessed by the task during the execution of that instruction. All in all, seven listeners and two factories are replaced in JPF consisting of roughly 1.6K lines of code.

The approach in this paper is compared to two other approaches implemented by JPF: *Precise race detector* (PRD) and *Gradual permission regions* (GPR). The PRD algorithm is a partial order reduction based on JPF’s ability to dynamically detect shared memory accesses. In this mode, JPF schedules on all detected shared memory accesses. GPR uses program annotations to reduce the number of shared locations that need to consider scheduling by grouping several bytecodes that access shared locations into a single atomic block of code with read/write indications [22]. For example, if there are two bytecodes that touch shared memory locations, PRD schedules from each of the two locations. In contrast, if those two locations are wrapped in a single permission region, then GPR only considers schedules from the start of the region with the region being considered atomic. GPR is equal to PRD if every bytecode that accesses shared memory is put in its own region. Both approaches are a form of partial order reduction with GPR outperforming PRD by virtue of considering significantly fewer scheduling points via the user annotated permission regions.

The comparison over a set of benchmarks is shown in Table 1. The benchmarks are a collection of those from the HJ distribution itself<sup>1</sup> and various presentation materials introducing the Habanero model; other benchmarks come from testing various language constructs in the development process. The table indicates for each benchmark its relative size in lines-of-code and tasks. The number of states generated by JPF for the proof, the time in minutes and sec-

<sup>1</sup> <https://wiki.rice.edu/confluence/display/HABANERO/Habanero-Java>.

**Table 1.** Computation graphs vs permission regions vs. PreciseRaceDetector.

Test ID	SLOC	Tasks	Computation graphs			GPR			PRD		
			States	mm:ss	Race	States	mm:ss	Race	States	mm:ss	Race
<i>Primitive Array Race</i>	39	3	5	00:00	Y	5	00:00	Y	220	00:00	Y
<i>Reciprocal Array Sum</i>	58	2	4	00:08	Y	32	00:06	Y	-	-	-
<i>Primitive Array No Race</i>	29	3	5	00:00	N	5	00:00	N	11,852	00:00	N
<i>Two Dim Arrays</i>	30	11	15	00:00	N	15	00:00	N	597	00:00	Y*
<i>ForAll With Iterable</i>	38	2	9	00:00	N	9	00:00	N	-	-	-
<i>Integer Counter Isolated</i>	54	10	24	00:01	N	1,013,102	05:53	N	-	-	-
<i>Pipeline With Futures</i>	69	5	34	00:00	N	34	00:00	N	-	-	-
<i>Prime Num Counter</i>	51	25	776	00:01	N	3,542,569	17:37	N	-	-	-
<i>Prime Num Counter ForAll</i>	52	25	30	00:02	N	18	00:01	N	-	-	-
<i>Prime Num Counter ForAsync</i>	44	11	653	00:01	N	2,528,064	15:44	N	-	-	-
<i>Add</i>	67	3	11	00:01	N	62,374	00:33	N	4930	00:03	Y*
<i>Scalar Multiply</i>	55	3	15	00:01	N	55,712	00:30	N	826	00:01	Y*
<i>Vector Add</i>	50	3	5	00:00	N	17	00:00	N	46,394	00:19	N
<i>Clumped Access</i>	30	3	5	00:03	N	15	00:00	N	-	-	-

onds, and finally whether or not a race was detected. The “-” indicates that no results are available because the approach exceeded the arbitrary one hour time bound for each run. The experiments were run on a machine with an Intel Core i5 processor with 2.6 GHz speed and 8 GB of RAM.

The table shows that in general, PRD does not finish in the time bound. The “Y\*” on the **Race** column for PRD indicates that PRD incorrectly reports data-race on array objects in some examples because it does not check the indexes—a shortcoming in the PRD implementation. GPR falls behind quickly as the number of permission regions grow. The difference in performance is seen in the *Add*, *Scalar multiply*, and *Prime number counter* benchmarks which use shared variables. The regions are made as big as possible without creating a data-race. The *Prime number counter* benchmark also has isolated sections. As a result, the state space for *computation graphs* is also large compared to other benchmarks. Of course, in the presence of isolation, the approach in this paper must enumerate all possible computation graphs, so it suffers the same state explosion as other model checking approaches.

The next set of results are for bigger *real-world* programs. The *Crypt-af* and *Crypt-f* benchmarks are implementations of the IDEA encryption algorithm and *Series-af* and *Series-f* are the Fourier coefficient analysis benchmarks adapted from the JGF suite [23] using **async-finish** and **future** constructs respectively. The *strassen* benchmark is adapted from the OpenMP version of the program in the Kastors suite [24]. These are quickly verified free of data-race using computation graphs as shown below—PRD and GPR time out. Source code and additional benchmarks converted from <https://github.com/LLNL/dataracebench> can be found at <https://jpf.byu.edu/jpf-hj/>.

Test ID	SLOC	Tasks	States	mm:ss	Race
<i>Crypt-af</i>	1010	259	260	00:17	N
<i>Crypt-f</i>	1145	387	775	00:46	N
<i>Series-af</i>	730	329	750	00:36	N
<i>Series-f</i>	830	354	630	00:51	N
<i>Strassen</i>	560	3	7	00:57	N

## 6 Related Work

Data-race detection in *unstructured thread parallelism*, where there is no defined protocol for creating and joining threads, or accessing shared memory, relies on static analysis to approximate parallelism and memory accesses [25–27] and then improves precision with dynamic analysis [17, 28–31]. Other approaches reason about threads individually [32, 33]. These approaches make few assumptions about the parallelism for generality and typically have higher cost for analysis. It is difficult to compare the approach in this paper to these more general approaches because the work in this paper relies critically on the structure of the parallelism to reduce the cost of formal analysis.

*Structured parallelism* constrains how threads are created and joined and how shared memory is accessed through programming models. For example, a locking protocol leads to static, dynamic, or hybrid lock-set analyses for data-race detection that are typically more efficient than approaches to unstructured parallelism [34–36]. Locking protocols can be applied to isolation with similar results—over-approximating the set of shared locations potentially rejecting programs as having data-race when indeed they do not.

Dynamic data-race detection based on *SP-bags* has been shown to effectively scale to large program instances and the method has been applied to the Habanero programming model to support a limited set of Habanero keywords including futures but not isolation [10]. The goal in this paper is verification and not run time monitoring, so it needs to enumerate all possible computation graphs but can benefit from the more efficient SP-bags algorithm to detect data-race on-the-fly in the computation graph.

Programmer annotations indicating shared interactions (e.g., permission regions) do improve model checking in general [13]. These are best understood as helping the partial order reduction by grouping several shared accesses into a single atomic block. The regions are then annotated with read/write properties to indicate what the atomic block is doing. The model checker only considers the interactions of these shared regions to reduce the number of executions explored to prove the system correct.

There are other model checkers for task parallel languages [14, 15]. The first modifies JPF and an X10 run-time extensively (beyond the normal JPF options for customization) and the second is a new virtual machine to model check the language. Both of these solutions require extensive programming whereas the solution in this paper leverages the existing Habanero verification runtime for

JPF. That run-time maps tasks to threads making it small enough (relatively few lines of code) to argue correctness and making it work with JPF without any modification to JPF internals.

## 7 Conclusion and Future Work

This paper presents a model checking approach for data race detection in task parallel programs using computation graphs. The computation graph represents the happens-before relation of the task parallel program and can readily be checked for data-race. The approach then enumerates all computation graphs created by different schedules of isolated regions to prove data-race freedom. The data race detection analysis is implemented for a Java implementation of the Habanero programming model using JPF and evaluated on a host of benchmarks. The results are compared to JPF's precise race detector and a gradual permission regions based extension. The results show that computation graph analysis reduces the time required for verification significantly relative to JPF's standards.

Future work is to reduce the number of schedules that must be considered by the model checker by weakening the happens-before relation in a manner similar to recent advances in dynamic data-race detection [18, 19]. Soundly weakening the happens-before relation grows the number of schedules covered by any one observed program execution including schedules that have different orders of isolation statements. These larger equivalence classes captured by the weakened happens-before relation can be used to prune schedules from consideration by the model checker. Other future work is to leverage static analysis, abstract interpretation, to reason over the input space so that the proof can be generalized to all inputs and executions.

**Acknowledgement.** This work is supported by the National Science Foundation under grant 1302524. Thanks to Lincoln Bergeson and Kyle Storey for their contribution to the tool and its tests.

## References

1. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. *J. Parallel Distrib. Comput.* **37**(1), 55–69 (1996)
2. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.* **40**(10), 519–538 (2005)
3. Cavé, V., Zhao, J., Shirako, J., Sarkar, V.: Habanero-Java: the new adventures of old X10. In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pp. 51–61. ACM (2011)
4. Imam, S., Sarkar, V.: Habanero-Java library: a Java 8 framework for multicore programming. In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pp. 75–86. ACM (2014)



5. Feng, M., Leiserson, C.E.: Efficient detection of determinacy races in Cilk programs. In: Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1997, pp. 1–11. ACM, New York (1997)
6. Cheng, G.I., Feng, M., Leiserson, C.E., Randall, K.H., Stark, A.F.: Detecting data races in Cilk programs that use locks. In: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1998, pp. 298–309. ACM, New York (1998)
7. Bender, M.A., Fineman, J.T., Gilbert, S., Leiserson, C.E.: On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2004, pp. 133–144. ACM, New York (2004)
8. Raman, R., Zhao, J., Sarkar, V., Vechev, M.T., Yahav, E.: Efficient data race detection for async-finish parallelism. *Form. Methods Syst. Des.* **41**(3), 321–347 (2012)
9. Utterback, R., Agrawal, K., Fineman, J.T., Lee, I.T.A.: Provably good and practically efficient parallel race detection for fork-join programs. In: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, pp. 83–94. ACM, New York (2016)
10. Surendran, R., Sarkar, V.: Dynamic determinacy race detection for task parallelism with futures. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 368–385. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-46982-9\\_23](https://doi.org/10.1007/978-3-319-46982-9_23)
11. Mellor-Crummey, J.: On-the-fly detection of data races for programs with nested fork-join parallelism. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing 1991, pp. 24–33. ACM, New York (1991)
12. Raman, R., Zhao, J., Sarkar, V., Vechev, M., Yahav, E.: Scalable and precise dynamic datarace detection for structured parallelism. In: ACM SIGPLAN Notices, vol. 47, no. 6, pp. 531–542. ACM (2012)
13. Westbrook, E., Zhao, J., Budimčić, Z., Sarkar, V.: Practical permissions for race-free parallelism. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 614–639. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-31057-7\\_27](https://doi.org/10.1007/978-3-642-31057-7_27)
14. Gligoric, M., Mehrlitz, P.C., Marinov, D.: X10X: model checking a new programming language with an “old” model checker. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pp. 11–20. IEEE (2012)
15. Zirkel, T.K., Siegel, S.F., McClory, T.: Automated verification of chapel programs using model checking and symbolic execution. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 198–212. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-38088-4\\_14](https://doi.org/10.1007/978-3-642-38088-4_14)
16. Anderson, P., Chase, B., Mercer, E.: JPF verification of Habanero Java programs. *ACM SIGSOFT Softw. Eng. Notes* **39**(1), 1–7 (2014)
17. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21**(7), 558–565 (1978)
18. Kini, D., Mathur, U., Viswanathan, M.: Dynamic race prediction in linear time. *SIGPLAN Not.* **52**(6), 157–170 (2017)
19. Huang, J., Meredith, P.O., Rosu, G.: Maximal sound predictive race detection with control flow abstraction. *SIGPLAN Not.* **49**(6), 337–348 (2014)
20. Dennis, J.B., Gao, G.R., Sarkar, V.: Determinacy and repeatability of parallel program schemata. In: 2012 Data-Flow Execution Models for Extreme Scale Computing (DFM), pp. 1–9. IEEE (2012)
21. Bouajjani, A., Emmi, M.: Analysis of recursively parallel programs. *ACM SIGPLAN Not.* **47**(1), 203–214 (2012)

22. Mercer, E., Anderson, P., Vrvilo, N., Sarkar, V.: Model checking task parallel programs using gradual permissions. In: Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering, New Ideas Category, pp. 535–540. ACM (2015)
23. Bull, J.M., Smith, L.A., Westhead, M.D., Henty, D.S., Davey, R.A.: A benchmark suite for high performance Java. *Concurr. - Pract. Exp.* **12**(6), 375–388 (2000)
24. Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., Gautier, T.: Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 16–29. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-11454-5\\_2](https://doi.org/10.1007/978-3-319-11454-5_2)
25. Kahlon, V., Sinha, N., Kruus, E., Zhang, Y.: Static data race detection for concurrent programs with asynchronous calls. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the Foundations of Software Engineering, pp. 13–22. ACM (2009)
26. Kulikov, S., Shafiei, N., Van Breugel, F., Visser, W.: Detecting data races with Java PathFinder (2010)
27. Vechev, M., Yahav, E., Raman, R., Sarkar, V.: Automatic verification of determinism for structured parallel programs. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 455–471. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-15769-1\\_28](https://doi.org/10.1007/978-3-642-15769-1_28)
28. Godefroid, P.: Model checking for programming languages using Verisoft. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1997, pp. 174–186. ACM, New York (1997)
29. Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: ACM SIGPLAN Notices, vol. 44, no. 6, pp. 121–133. ACM (2009)
30. Choi, J.D., Lee, K., Loginov, A., O’Callahan, R., Sarkar, V., Sridharan, M.: Efficient and precise datarace detection for multithreaded object-oriented programs. *SIGPLAN Not.* **37**(5), 258–269 (2002)
31. Dimitrov, D., Raychev, V., Vechev, M., Koskinen, E.: Commutativity race detection. In: ACM SIGPLAN Notices, vol. 49, no. 6, pp. 305–315. ACM (2014)
32. Malkis, A., Podelski, A., Rybalchenko, A.: Precise thread-modular verification. In: Nielson, H.R., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 218–232. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-74061-2\\_14](https://doi.org/10.1007/978-3-540-74061-2_14)
33. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: ACM SIGPLAN Notices, vol. 42, no. 6, pp. 266–277. ACM (2007)
34. Engler, D., Ashcraft, K.: RacerX: effective, static detection of race conditions and deadlocks. In: ACM SIGOPS Operating Systems Review, vol. 37, no. 5, pp. 237–252. ACM (2003)
35. Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: a race and transaction-aware Java runtime. In: ACM SIGPLAN Notices, vol. 42, no. 6, pp. 245–255. ACM (2007)
36. Voung, J.W., Jhala, R., Lerner, S.: RELAY: static race detection on millions of lines of code. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 205–214. ACM (2007)



# Consistency of Property Specification Patterns with Boolean and Constrained Numerical Signals

Massimo Narizzano<sup>1</sup> , Luca Pulina<sup>2</sup> , Armando Tacchella<sup>1</sup> ,  
and Simone Vuotto<sup>1,2</sup> 

<sup>1</sup> DIBRIS, University of Genoa, Viale Causa 13, 16145 Genoa, Italy  
{massimo.narizzano, armando.tacchella}@unige.it

<sup>2</sup> Chemistry and Pharmacy Department, University of Sassari,  
Via Vienna 2, Sassari, Italy  
{lpulina, svuotto}@uniss.it

**Abstract.** Property Specification Patterns (PSPs) have been proposed to solve recurring specification needs, to ease the formalization of requirements, and enable automated verification thereof. In this paper, we extend PSPs by considering Boolean as well as atomic numerical assertions. This extension enables us to reason about functional requirements which would not be captured by basic PSPs. We contribute an encoding from constrained PSPs to LTL formulae, and we show experimental results demonstrating that our approach scales on requirements of realistic size generated using a probabilistic model. Finally, we show that our extension enables us to prove (in)consistency of requirements about an embedded controller for a robotic manipulator.

## 1 Introduction

In the context of safety- and security-critical cyber-physical systems (CPSs), checking the consistency of functional requirements is an indisputable, yet challenging task. Requirements written in natural language call for time-consuming and error-prone manual reviews, whereas enabling automated consistency verification often requires overburdening formalizations. Given the increasing pervasiveness of CPSs, their stringent time-to-market and product budget constraints, practical solutions to enable automated verification of requirements are in order, and Property Specification Patterns (PSPs) [8] offer a viable path towards this target. PSPs are a collection of parameterizable, high-level, formalism-independent specification abstractions, originally developed to capture recurring solutions to the needs of requirement engineering. Each pattern can be directly encoded in a formal specification language, such as linear time temporal logic (LTL) [18], computational tree logic (CTL) [2], or graphical interval logic (GIL) [5]. Because of their features, PSPs may ease the burden of formalizing requirements, yet enable their verification using current state-of-the-art automated reasoning tools—see, e.g., [1, 9, 11, 14, 23].

The original formulation of PSPs caters for temporal structure over Boolean variables. However, for most practical applications, such expressiveness is too restricted. This is the case of the embedded controller for robotic manipulators that is under development in the context of the EU project CERBERO<sup>1</sup> and provides the main motivation for this work. As an example, consider the following statement: “*The angle of joint1 shall never be greater than 170 degrees*”. This requirement imposes a safety threshold related to some joint of the manipulator (*joint1*) with respect to physically-realizable poses, yet it cannot be expressed as a PSP unless we add atomic numerical assertions in a constraint system  $\mathcal{D}$ . We call Constraint PSP, or  $\text{PSP}(\mathcal{D})$  for short, a pattern which has the same structure of a PSP, but contains atomic propositions from  $\mathcal{D}$ . For instance, using  $\text{PSP}(\mathbb{R}, <, =)$  we can rewrite the above requirement as an *universality* pattern: “*Globally, it is always the case that  $\theta_1 < 170$  holds*”, where  $\theta_1$  is the numerical signal (variable) for the angle of *joint1*. In principle, automated reasoning about Constraint PSPs can be performed in Constraint Linear Temporal Logic, i.e., LTL extended with atomic assertions from a constraint system [4]: in our example above, the encoding would be simply  $\Box(\theta_1 < 170)$ . Unfortunately, this approach does not always lend itself to a practical solution, because Constraint Linear Temporal Logic is undecidable in general [3]. Restrictions on  $\mathcal{D}$  may restore decidability [4], but they introduce limitations in the expressiveness of the corresponding PSPs.

In this paper, we propose a solution which ensures that automated verification of requirements is feasible, yet enables PSPs mixing both Boolean variables and (constrained) numerical signals. Our approach enables us to capture many specifications of practical interest, and to pick a verification procedure from the relatively large pool of automated reasoning systems currently available for LTL. In particular, we restrict our attention to a constraint systems of the form  $(\mathbb{R}, <, =)$ , and atomic propositions of the form  $x < C$  or  $x = C$ , where  $x \in \mathbb{R}$  is a variable and  $C \in \mathbb{R}$  is a constant value. In the following, we write  $\mathcal{D}_C$  to denote such restriction. Our contribution can be summarized as follows:

- We extend basic PSPs over the constraint system  $\mathcal{D}_C$ , and we provide an encoding from any  $\text{PSP}(\mathcal{D}_C)$  into a corresponding LTL formula.
- We provide a tool<sup>2</sup> based on state-of-the-art decision procedures and model checkers to automatically analyze requirements expressed as  $\text{PSPs}(\mathcal{D}_C)$ .
- We implement a generator of artificial requirements expressed as  $\text{PSPs}(\mathcal{D}_C)$ ; the generator takes a set of parameters in input and emits a collection of PSPs according to a parametrized probability model.
- Using our generator, we run an extensive experimental evaluation aimed at understanding (i) which automated reasoning tool is best at handling set of requirements as  $\text{PSPs}(\mathcal{D}_C)$ , and (ii) whether our approach is scalable.
- Finally, we analyze the requirements of the aforementioned embedded controller, experimenting also with the addition of faulty ones.

<sup>1</sup> Cross-layer modEl-based fRamework for multi-oBjective dEsign of Reconfigurable systems in unceRtain hybRid enviroNments—<http://www.cerbero-h2020.eu/>.

<sup>2</sup> <https://github.com/SAGE-Lab/snl2fl>.

The consistency of requirements written in  $\text{PSP}(\mathcal{D}_C)$  is carried out using tools and techniques available in the literature [11, 21, 22]. With those, we demonstrate the scalability of our approach by checking the consistency of up to 1920 requirements, featuring 160 variables and up to 8 thresholds appearing in the atomic assertions, within less than 500 CPU seconds. A total of 75 requirements about the embedded controller for the CERBERO project is checked in a matter of seconds, even without resorting to the best tool among those we consider.

The rest of the paper is organized as follows. Section 2 contains some basic concepts on LTL, PSPs and some related work. In Sect. 3 we present the extension of basic PSPs over  $\mathcal{D}_C$  and the related encoding to LTL. In Sects. 4 and 5 we report the results of the experimental analysis concerning the scalability and the case study on the embedded controller, respectively. We conclude the paper in Sect. 6 with some final remarks.

## 2 Background and Related Work

*LTL syntax and semantics.* Linear temporal logic (LTL) [17] formulae are built on a finite set  $Prop$  of atomic propositions as follows:

$$\phi = p \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \mathcal{X}\phi_1 \mid \phi_1 \mathcal{U}\phi_2$$

where  $p \in Prop$ ,  $\phi, \phi_1, \phi_2$  are LTL formulae,  $\mathcal{X}$  is the “next” operator and  $\mathcal{U}$  is the “until” operator. An LTL formula is interpreted over a *computation*, i.e., a function  $\pi : \mathbb{N} \rightarrow 2^{Prop}$  which assigns truth values to the elements of  $Prop$  at each time instant (natural number). For a computation  $\pi$  and a point  $i \in \mathbb{N}$ :

- $\pi, i \models p$  for  $p \in Prop$  iff  $p \in \pi(i)$
- $\pi, i \models \neg\alpha$  iff  $\pi, i \not\models \alpha$
- $\pi, i \models (\alpha \vee \beta)$  iff  $\pi, i \models \alpha$  or  $\pi, i \models \beta$
- $\pi, i \models \mathcal{X}\alpha$  iff  $\pi, i + 1 \models \alpha$
- $\pi, i \models \alpha \mathcal{U}\beta$  iff for some  $j \geq i$ , we have  $\pi, j \models \beta$  and for all  $k, i \leq k < j$  we have  $\pi, k \models \alpha$

We say that  $\pi$  *satisfies* a formula  $\phi$ , denoted  $\pi \models \phi$ , iff  $\pi, 0 \models \phi$ . If  $\pi \models \phi$  for every  $\pi$ , then  $\phi$  is *true* and we write  $\models \phi$ . We abbreviate  $p \vee \neg p$  as  $\top$ ,  $p \wedge \neg p$  as  $\perp$  and we consider other Boolean connectives like “ $\wedge$ ” and “ $\rightarrow$ ” with the usual meaning. We introduce  $\diamond\phi$  (“eventually”) to denote  $\top \mathcal{U}\phi$  and  $\square\phi$  (“always”) to denote  $\neg\diamond\neg\phi$ . Finally, some of the PSPs use the “weak until” operator defined as  $\alpha \mathcal{W}\beta = \square\alpha \vee (\alpha \mathcal{U}\beta)$ .

*LTL satisfiability.* Among various approaches to decide LTL satisfiability, reduction to model checking was proposed in [20] to check the consistency of requirements expressed as LTL formulae. Given a formula  $\phi$  over a set  $Prop$  of atomic propositions, a *universal* model  $M$  can be constructed. Intuitively, a universal model encodes all the possible computations over  $Prop$  as (infinite) traces, and therefore  $\phi$  is satisfiable precisely when  $M$  does not satisfy  $\neg\phi$ . In [22] a first improvement over this basic strategy is presented together with the tool PANDA<sup>3</sup> whereas in [13] an algorithm based on automata construction is

<sup>3</sup> <https://ti.arc.nasa.gov/m/profile/kyrozier/PANDA/PANDA.html>.

proposed to enhance performances even further—the approach is implemented in a tool called AALTA. Further studies along this direction include [11, 12]. In the latter, a portfolio LTL satisfiability solver called POLSAT is proposed to run different techniques in parallel and return the result of the first one to terminate successfully.

<b>Response</b>	
Describe cause-effect relationships between a pair of events/states. An occurrence of the first, the cause, must be followed by an occurrence of the second, the effect. Also known as Follows and Leads-to.	
<hr/>	
<b>Structured English Grammar</b>	
<i>It is always the case that if P holds, then S eventually holds.</i>	
<hr/>	
<b>LTL Mappings</b>	
Globally	$\Box (P \rightarrow \Diamond S)$
Before R	$\Diamond R \rightarrow (P \rightarrow (\bar{R} \mathcal{U} (S \wedge \bar{R}))) \mathcal{U} R$
After Q	$\Box (Q \rightarrow \Box (P \rightarrow \Diamond S))$
Between Q and R	$\Box ((Q \wedge \bar{R} \wedge \Diamond R) \rightarrow (P \rightarrow (\bar{R} \mathcal{U} (S \wedge \bar{R}))) \mathcal{U} R)$
After Q until R	$\Box (Q \wedge \bar{R} \rightarrow ((P \rightarrow (\bar{R} \mathcal{U} (S \wedge \bar{R}))) \mathcal{W} R))$
<hr/>	
<b>Example</b>	
<i>If the train is approaching, then the gate shall be closed.</i>	

**Fig. 1.** Response pattern ( $\bar{\alpha}$  stands for  $\neg\alpha$ ).

*Property Specification Patterns (PSPs).* The original proposal of PSPs is to be found in [8]. They are meant to describe the essential structure of system’s behaviours and provide expressions of such behaviors in a range of common formalisms. An example of a PSP is given in Fig. 1—with some part omitted for sake of readability.<sup>4</sup> A pattern is comprised of a *Name* (Response in Fig. 1), an (informal) statement describing the behaviour captured by the pattern, and a (structured English) statement [10] that should be used to express requirements. The LTL mappings corresponding to different declinations of the pattern are also given, where capital letters ( $P, S, T$ , etc.) stands for Boolean states/events.<sup>5</sup> In more detail, a PSP is composed of two parts: (i) the *scope*, and (ii) the *body*. The *scope* is the extent of the program execution over which the pattern must hold,

<sup>4</sup> The full list of PSPs considered in this paper and their mapping to LTL and other logics is available at <http://patterns.projects.cis.ksu.edu/>.

<sup>5</sup> We omitted some aspects which are not relevant for our work, e.g., translations to other logics like CTL [8].

and there are five scopes allowed: *Globally*, to span the entire scope execution; *Before*, to span execution up to a state/event; *After*, to span execution after a state/event; *Between*, to cover the part of execution from one state/event to another one; *After-until*, where the first part of the pattern continues even if the second state/event never happens. For state-delimited scopes, the interval in which the property is evaluated is closed at the left and open at the right end. The *body* of a pattern, describes the behavior that we want to specify. In [8] the bodies are categorized in *occurrence* and *order* patterns. Occurrence patterns require states/events to occur or not to occur. Examples of such bodies are *Absence*, where a given state/event must not occur within a scope, and its opposite *Existence*. Order patterns constrain the order of the states/events. Examples of such patterns are *Precedence*, where a state/event must always precede another state/event, and *Response*, where a state/event must always be followed by another state/event within the scope. Moreover, we included the *Invariant* pattern introduced in [19], and dictating that a state/event must occur whenever another state/event occurs. Combining scopes and bodies we can construct 55 different types of patterns.

*Related Work.* In [15] the framework, *Property Specification Pattern Wizard (PSP-Wizard)* is presented, for machine-assisted definition of temporal formulae capturing pattern-based system properties. PSP-Wizard offers a translation into LTL of the patterns encoded in the tool, but it is meant to aid specification, rather than support consistency checking, and it cannot deal with numerical signals. In [10], an extension is presented to deal with real-time specifications, together with mappings to Metric temporal logic (MTL), Timed computational tree logic (TCTL) and Real-time graphical interval logic (RTGIL). Even if this work is not directly connected with ours, it is worth mentioning it since their structured English grammar for patterns is at the basis of our formalism. The work in [10] also provided inspiration to a recent set of works [6,7] about a tool, called VI-Spec, to assist the analyst in the elicitation and debugging of formal specifications. VI-Spec lets the user specify requirements through a graphical user interface, translates them to MITL formulae and then supports debugging of the specification using run-time verification techniques. VI-Spec embodies an approach similar to ours to deal with numerical signals by translating inequalities to sets of Boolean variables. However, VI-Spec differs from our work in several aspects, most notably the fact that it performs debugging rather than consistency, so the behavior of each signal over time must be known. Also, VI-Spec handles only inequalities and does not deal with sets of requirements written using PSPs.

### 3 Constraint Property Specification Patterns

Let us start by defining a *constraint system*  $\mathcal{D}$  as a tuple  $\mathcal{D} = (D, R_1, \dots, R_n, \mathcal{I})$ , where  $D$  is a non-empty set called *domain*, and each  $R_i$  is a predicate symbol of arity  $a_i$ , with  $\mathcal{I}(R_i) \subseteq D^{a_i}$  being its interpretation. Given a set of variables

$X$  and a set of constants  $C$  such that  $C \cap X = \emptyset$ , a *term* is a member of the set  $T = C \cup X$ ; an (atomic)  $\mathcal{D}$ -*constraint* over a set of terms is of the form  $R_i(t_1, \dots, t_{a_i})$  for some  $1 \leq i \leq n$  and  $t_j \in T$  for all  $1 \leq j \leq a_i$ —we also use the term *constraint* when  $\mathcal{D}$  is understood from the context. We define *linear temporal logic modulo constraints*—LTL( $\mathcal{D}$ ) for short—as an extension of LTL with atoms in a constraint system  $\mathcal{D}$ . Given a set of Boolean propositions  $Prop$ , a constraint system  $\mathcal{D} = (D, R_1, \dots, R_n, \mathcal{I})$ , and a set of terms  $T = C \cup X$ , an LTL( $\mathcal{D}$ ) formula is defined as:

$$\phi = p \mid R_i(t_1, \dots, t_{a_i}) \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \mathcal{X} \phi_1 \mid \phi_1 \mathcal{U} \phi_2$$

where  $p \in Prop$ ,  $\phi, \phi_1, \phi_2$  are LTL( $\mathcal{D}$ ) formulas, and  $R_i(\cdot)$  with  $1 \leq i \leq n$  is an atomic  $\mathcal{D}$ -constraint. Additional Boolean and temporal operators are defined as in LTL with the same intended meaning. Notice that the set of LTL( $\mathcal{D}$ ) formulas is a (strict) subset of those in constraint linear temporal logic—CLTL( $\mathcal{D}$ ) for short—as defined, e.g., in [4]. LTL( $\mathcal{D}$ ) formulas are interpreted over computations of the form  $\pi : \mathbb{N} \rightarrow 2^{Prop}$  plus additional *evaluations* of the form  $\nu : T \times \mathbb{N} \rightarrow D$  such that, for all  $i \in \mathbb{N}$ ,  $\nu(c, i) = \nu(c) \in D$  for all  $c \in C$ , whereas  $\nu(x, i) \in D$  for all  $x \in X$ . In words, the function  $\nu$  associates to constants  $c \in C$  a value  $\nu(c)$  that does not change in time, and to variables  $x \in X$  a value  $\nu(x, i)$  that possibly changes at each time instant  $i \in \mathbb{N}$ . LTL semantics is extended to LTL( $\mathcal{D}$ ) by handling constraints:

$$\pi, \nu, j \models R_i(t_1, \dots, t_{a_i}) \text{ iff } (\nu(t_1, j), \dots, \nu(t_{a_i}, j)) \in \mathcal{I}(R_i)$$

We say that  $\pi$  and  $\nu$  *satisfy* a formula  $\phi$ , denoted  $\pi, \nu \models \phi$ , iff  $\pi, \nu, 0 \models \phi$ . A formula  $\phi$  is *satisfiable* as long as there exist a computation  $\pi$  and a valuation  $\nu$  such that  $\pi, \nu \models \phi$ . We further restrict our attention to the constraint system  $\mathcal{D}_C = (\mathbb{R}, <, =)$ , with atomic constraints of the form  $x < c$  and  $x = c$ , where  $c$  is a constant corresponding to some real number—we abuse notation and write  $c \in \mathbb{R}$ —and the interpretation of the predicates “ $<$ ” and “ $=$ ” is the usual one. While CLTL( $\mathcal{D}$ ) is undecidable in general [3, 4], LTL( $\mathcal{D}_C$ ) is decidable since, as we show in the following, it can be reduced to LTL satisfiability.

We introduce the concept of *constraint property specification pattern*, denoted PSP( $\mathcal{D}$ ), to deal with specifications containing Boolean variables as well as atoms from a constraint system  $\mathcal{D}$ . In particular, a PSP( $\mathcal{D}_C$ ) features only Boolean atoms and atomic constraints of the form  $x < c$  or  $x = c$  ( $c \in \mathbb{R}$ ). For example, the requirement:

*The angle of joint1 shall never be greater than 170 degrees*

can be re-written as a PSP( $\mathcal{D}_C$ ):

*Globally, it is always the case that  $\theta_1 < 170$*

where  $\theta_1 \in \mathbb{R}$  is the variable associated to the angle of *joint1* and 170 is the limiting threshold. While basic PSPs only allow for Boolean states/events in their description, PSPs( $\mathcal{D}_C$ ) also allow for atomic numerical constraints. It is



straightforward to extend the translation of [8] from basic PSPs to LTL in order to encode any PSP( $\mathcal{D}_C$ ) to a formula in LTL( $\mathcal{D}_C$ ). Consider, for instance, the set of requirements:

$R_1$  Globally, it is always the case that  $\mathbf{v} \leq \mathbf{5.0}$  holds.

$R_2$  After  $\mathbf{a}$ ,  $\mathbf{v} \leq \mathbf{8.5}$  eventually holds.

$R_3$  After  $\mathbf{a}$ , it is always the case that if  $\mathbf{v} \geq \mathbf{3.2}$  holds, then  $\mathbf{z}$  eventually holds.

where  $\mathbf{a}$  and  $\mathbf{z}$  are Boolean states/events, whereas  $\mathbf{v}$  is a numeric signal. These PSPs( $\mathcal{D}_C$ )<sup>6</sup> can be rewritten as the following LTL( $\mathcal{D}_C$ ) formula:

$$\begin{aligned} & \Box(v < 5.0 \vee v = 5.0) \quad \wedge \\ & \Box(a \rightarrow \Diamond(v < 8.5) \vee (v = 8.5)) \wedge \\ & \Box(a \rightarrow \Box(\neg(v < 3.2) \rightarrow \Diamond z)) \end{aligned} \quad (1)$$

Therefore, to reason about the consistency of sets of requirements written using PSPs( $\mathcal{D}_C$ ) it is sufficient to provide an algorithm for deciding the satisfiability of LTL( $\mathcal{D}_C$ ) formulas.

To this end, consider an LTL( $\mathcal{D}_C$ ) formula  $\phi$ , and let  $X(\phi)$  be the set of variables and  $C(\phi)$  be the set of constants that occur in  $\phi$ . We define the *set of thresholds*  $S_x(\phi) \subseteq C(\phi)$  as the set of constant values against which variable  $x \in X(\phi)$  is compared to. More precisely, for every variable  $x \in X(\phi)$  we construct a set  $S_x(\phi) = \{c_1, \dots, c_n\}$  such that, for all  $c_k \in \mathbb{R}$  with  $1 \leq k \leq n$ ,  $\phi$  contains a constraint of the form  $x < c_k$  or  $x = c_k$ . In the following, for our convenience, we consider each threshold set  $S_x(\phi)$  ordered in ascending order, i.e.,  $c_k < c_{k+1}$  for all  $1 \leq k < n$ . For instance, in example (1), we have  $X = \{v\}$  and the set  $S_v = \{3.2, 5.0, 8.5\}$ . Given an LTL( $\mathcal{D}$ ) formula  $\phi$ , let  $S_x(\phi) = \{c_1, \dots, c_n\}$  be the ordered set of thresholds for some variable  $x \in X(\phi)$ ; given a computation  $\pi$  and a valuation  $\nu$  we can define:

- $Q_x(\phi) = \{q_1, \dots, q_n\}$  as the set of Boolean propositions such that, for  $1 < j \leq n$ , we have  $q_j \in \pi(i)$  for some  $i = 0, 1, \dots$  exactly when  $c_{j-1} < \nu(x, i) < c_j$ , and for  $j = 1$ , we have  $q_j \in \pi(i)$  for some  $i = 0, 1, \dots$  exactly when  $\nu(x, i) < c_j$ .
- $E_x(\phi) = \{e_1, \dots, e_n\}$  as the set of Boolean propositions such that we have  $e_j \in \pi(i)$  for  $i = 0, 1, \dots$  exactly when  $\nu(x, i) = c_j$ .

Notice that, by definition of  $Q_x(\phi)$  and  $E_x(\phi)$ , given any time instant  $i \in 0, 1, 2, \dots$ , we have that exactly one of the following cases is true ( $1 \leq j \leq n$ ):

- $q_j \in \pi(i)$  for some  $j$ ,  $q_l \notin \pi(i)$  for all  $l \neq j$  and  $e_j \notin \pi(i)$  for all  $j$ ;
- $e_j \in \pi(i)$  for some  $j$ ,  $e_l \notin \pi(i)$  for all  $l \neq j$  and  $q_j \notin \pi(i)$  for all  $j$ ;
- $q_j \notin \pi(i)$  and  $e_j \notin \pi(i)$  for all  $j$ .

<sup>6</sup> Strictly speaking, the syntax used is not that of  $\mathcal{D}_C$ , but a statement like  $v \leq 5.0$  can be thought as syntactic sugar for the expression  $(v < 5.0) \vee (v = 5.0)$ .

Intuitively, the first case above corresponds to a value of  $x$  that lies between some threshold value in  $S_x(\phi)$  or before its smallest value; the second case occurs when a threshold value is assigned to  $x$ , and the third case is when  $x$  exceeds the highest threshold value in  $S_x(\phi)$ . For instance, in example (1) we have  $S_v = \{3.2, 5.0, 8.5\}$  and the corresponding sets  $Q_v = \{q_1, q_2, q_3\}$  and  $E_v = \{e_1, e_2, e_3\}$ . Assuming, e.g.,  $\nu(v, i) = 10$  for some  $i = 0, 1, 2, \dots$ , we would have that  $Q_v \cap \pi(i) = E_v \cap \pi(i) = \emptyset$ .

Given the definitions above, an LTL( $\mathcal{D}$ ) formula  $\phi$  over the set of Boolean propositions  $Prop$  and the set of terms  $T = C \cup X$ , can be converted to an LTL formula  $\phi'$  over the set of Boolean propositions  $Prop \cup \bigcup_{\xi \in in X} (Q_\xi(\phi) \cup E_\xi(\phi))$ . We obtain this by considering, for each variable  $x \in X$  and associated threshold set  $S_x(\phi)$ , the corresponding propositions  $Q_x(\phi) = \{q_1, \dots, q_n\}$  and  $E_x = \{e_1, \dots, e_n\}$ ; then, for each  $t_k \in S_x(\phi)$ , we perform the following substitutions:

$$x < t_k \rightsquigarrow \bigvee_{j=1}^k q_j \vee \bigvee_{j=1}^{k-1} e_j \quad \text{and} \quad x = t_k \rightsquigarrow e_k. \tag{2}$$

However, replacing atomic numerical constraints is not enough to ensure equisatisfiability of  $\phi'$  with respect to  $\phi$ . In particular, we must encode the observation made above about “mutually exclusive” Boolean valuations for propositions in  $Q_x(\phi)$  and  $E_x(\phi)$  for every  $x \in X(\phi)$  as corresponding Boolean constraints:

$$\phi_M = \bigwedge_{\xi \in X(\phi)} \left( \bigwedge_{a, b \in M_\xi(\phi), a \neq b} \Box \neg(a \wedge b) \right) \tag{3}$$

where  $M_\xi(\phi) = Q_\xi(\phi) \cup E_\xi(\phi)$ . We can now state the following fact:

*Property 1.* Given an LTL( $\mathcal{D}_C$ ) formula  $\phi$  over the set of Boolean atoms  $Prop$  and the terms  $C(\phi) \cup X(\phi)$  we have that  $\phi$  is satisfiable if and only if the LTL formula  $\phi_M \rightarrow \phi'$  is satisfiable, where  $\phi'$  is obtained by replacing atomic numerical constraints according to rules (2) and  $\phi_M$  is defined according to (3).

For instance, given example (1), we have  $Q_v = \{q_1, q_2, q_3\}$  and  $E_v = \{e_1, e_2, e_3\}$  and the mutual exclusion constraints are written as:

$$\begin{aligned} \phi_M = & \Box \neg(q_1 \wedge q_2) \wedge \Box \neg(q_1 \wedge q_3) \wedge \Box \neg(q_1 \wedge e_1) \wedge \Box \neg(q_1 \wedge e_2) \wedge \\ & \Box \neg(q_1 \wedge e_3) \wedge \Box \neg(q_2 \wedge q_3) \wedge \Box \neg(q_2 \wedge e_1) \wedge \Box \neg(q_2 \wedge e_2) \wedge \\ & \Box \neg(q_2 \wedge e_3) \wedge \Box \neg(q_3 \wedge e_1) \wedge \Box \neg(q_3 \wedge e_2) \wedge \Box \neg(q_3 \wedge e_3) \wedge \\ & \Box \neg(e_1 \wedge e_2) \wedge \Box \neg(e_1 \wedge e_3) \wedge \Box \neg(e_2 \wedge e_3). \end{aligned} \tag{4}$$

Therefore, the LTL formula to be tested for assessing the consistency of the requirements is

$$\begin{aligned} \phi_M \rightarrow ( & \Box(q_1 \vee q_2 \vee e_1 \vee e_2) \wedge \\ & \Box(a \rightarrow \Diamond(\bigvee_{i=1}^3 q_i \vee e_i)) \wedge \\ & \Box(a \rightarrow \Box(\neg q_1 \rightarrow \Diamond z))). \end{aligned} \tag{5}$$

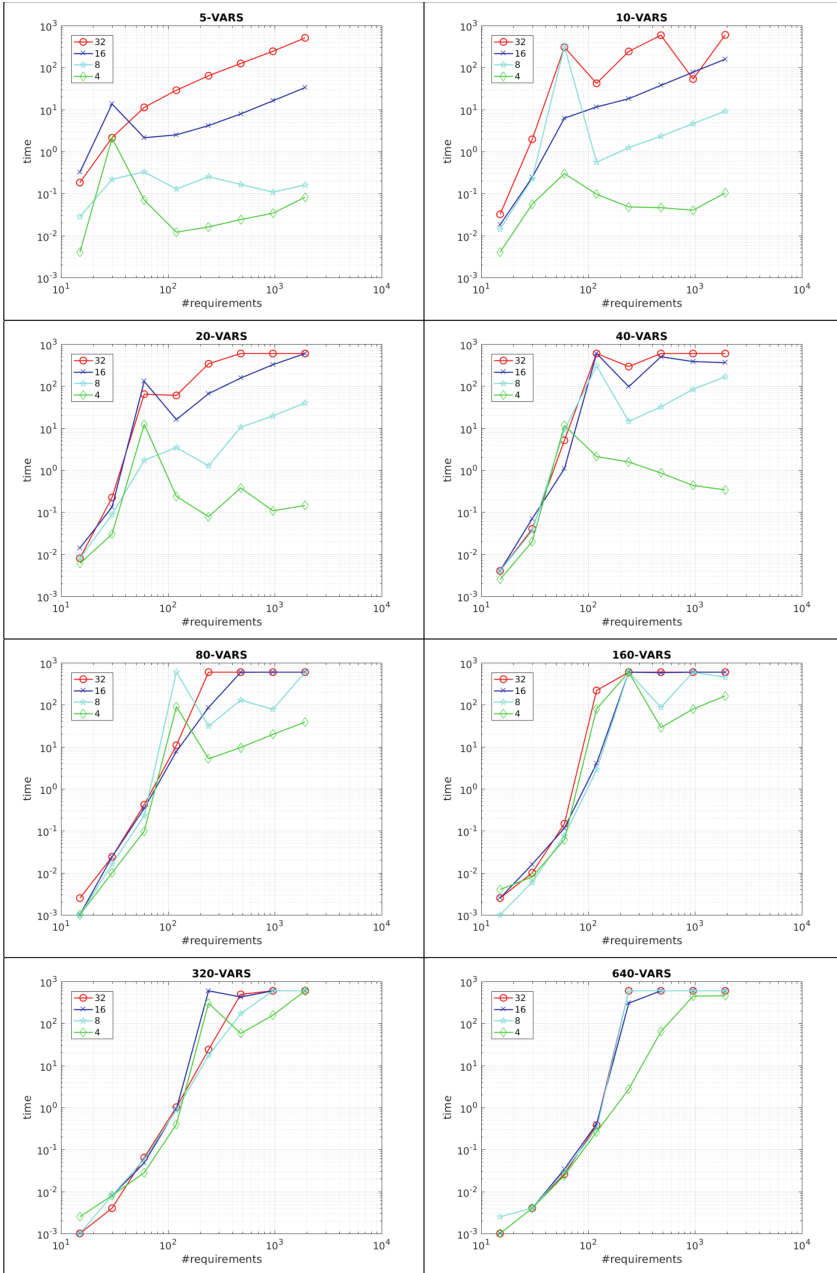
## 4 Analysis with Probabilistic Requirement Generation

The main goal of this Section is to investigate the scalability of our encoding from  $LTL(\mathcal{D})$  to  $LTL$ . To this end, we evaluate the performances<sup>7</sup> of some state-of-the-art tools for  $LTL$  satisfiability, and then we consider the best among such tools to assess whether our approach can scale to sets of requirements of realistic size. Since we want to have control over the kind of requirements, as well as the number of constraints and the size of the corresponding domains, we generate artificial specifications using a probabilistic model that we devised and implemented specifically to carry out the experiments herein presented. In particular, the following parameters can be tuned in our generator of specifications:

- The number of requirements generated ( $\#req$ ).
- The probability of each different body to occur in a pattern.
- The probability of each different scope to occur in a pattern.
- The size ( $\#vars$ ) of the set from which variables are picked uniformly at random to build patterns.
- The size ( $dom$ ) of the domain from which the thresholds of the atomic constraints are chosen uniformly at random.

*Evaluation of LTL satisfiability solvers.* The solvers considered in our analysis are the ones included in the portfolio solver POLSAT [11], namely AALTA [14], NUSMV [1], PLTL [23], and TRP++ [9]. In order to have a better understanding about the behavior of such solvers, we ran them separately instead of running POLSAT. Furthermore, in the case of NUSMV, we considered two different encodings. With reference to Property 1, the first encoding defines  $\phi_M$  as an invariant—denoted as NUSMV-INVAR—and  $\phi'$  is the property to check; the second encoding considers  $\phi_M \rightarrow \phi$  as the property to check—denoted as NUSMV-NOINVAR. In our experimental analysis we set the range of the parameters as follows:  $\#vars \in \{16, 32\}$ ,  $dom \in \{2, 4, 8, 16\}$ , and  $\#req \in \{8, 16, 32, 64\}$ . For each combination of the parameters with  $v \in \#vars$ ,  $r \in \#req$  and  $d \in dom$ , we generate 10 different benchmarks. Each benchmark is a specification containing  $r$  requirements where each scope has (uniform) probability 0.2 and each body has (uniform) probability 0.1. Then, for each atomic numerical constraint in the benchmark, we choose a variable out of  $v$  possible ones, and a threshold value out of  $d$  possible ones. In Table 1 we show the results of the analysis. Notice that we do not show the results of TRP++ because of the high number of failures obtained. Looking at the table, we can see that AALTA is the tool with the best performances, as it is capable of solving two times the problems solved by other solvers in most cases. Moreover, AALTA is up to 3 orders of magnitude faster than its competitors. Considering unsolved instances, it is worth noticing that in our experiments AALTA never reaches the granted time limit (10 CPU minutes), but

<sup>7</sup> All the experiments reported in this Section ran on a server equipped with 2 Intel Xeon E5-2640 v4 CPUs and 256 GB RAM running Debian with kernel 3.16.0-4.



**Fig. 2.** Scalability analysis. On the  $x$ -axes ( $y$ -axes resp.) we report  $\#req$  (CPU time in seconds resp.). Axis are both in logarithmic scale. In each plot we consider different values of  $\#dom$ . In particular, the diamond green line is for  $\#dom = 4$ , the light blue line with stars is for  $\#dom = 8$ , the blue crossed lines and red circled ones denote  $\#dom = 16$  and  $\#dom = 32$ , respectively. (Color figure online)

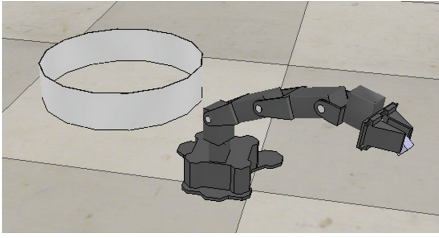
**Table 1.** Evaluation of LTL satisfiability solvers on randomly generated requirements. The first line reports the size of the domain (*dom*), while the second line reports the total amount of variables (*vars*) for each domain size. Then, for each tool (on the first column), the table shows the total amount of solved problems and the CPU time (in seconds) spent to solve them (columns “S” and “T”, respectively).

<i>dom</i>	2				4				8				16			
<i>#vars</i>	16		32		16		32		16		32		16		32	
Tool	S	T	S	T	S	T	S	T	S	T	S	T	S	T	S	T
AALTA	<b>16</b>	0.0	<b>27</b>	0.1	<b>22</b>	0.1	<b>29</b>	0.4	<b>26</b>	0.6	<b>29</b>	1.4	<b>25</b>	2.8	<b>31</b>	4.9
NUSMV-INVAR	11	30.4	10	185.1	10	804.2	9	881.3	11	68.1	8	402.9	10	1172.6	8	1001.9
NUSMV-NOINVAR	11	65.0	10	489.7	7	303.6	7	505.5	11	92.4	10	1277.6	8	660.0	9	1394.5
PLTL	8	25.0	11	108.1	9	1.2	10	0.6	10	19.6	11	0.1	11	14.5	14	3.5

it always fails beforehand. This is probably due to the fact that AALTA is still in a relatively early stage of development and it is not as mature as NUSMV and PLTL. Most importantly, we did not find any discrepancies in the satisfiability results of the evaluated tools.

*Evaluation of scalability.* The analysis involves 2560 different benchmarks generated as in the previous experiment. The initial value of *#req* has been set to 15, and it has been doubled until 1920, thus obtaining benchmarks with a total amount of requirements equals to 15, 30, 60, 120, 240, 480, 960, and 1920. Similarly has been done for *#vars* and *#dom*; the former ranges from 5 to 640, while the latter ranges from 4 to 32. At the end of the generation, we obtained 10 different sets composed of 256 benchmarks. In Fig. 2 we present the results, obtained running AALTA. The Figure is composed by 8 plots, one for each value of *#vars*. Looking at the plots in Fig. 2, we can see that the difficulty of the problem increases when all the values of the considered parameters increase, and this is particularly true considering the total amount of requirements. The parameter *#dom* has a higher impact of difficulty when the number of variables is small. Indeed, when the number of variables is less than 40 there is a clear difference between solving time with *#dom* = 4 and *#dom* = 32. On the other hand when the number of variables increases, all the plots for various values of *#dom* are very close to each other. As a final remark, we can see that even considering the largest problem (*#vars* = 640, *#dom* = 32), more than the 60% of the problems are solved by AALTA within the time limit of 10 min.

## 5 Analysis with a Controller for a Robotic Manipulator



**Fig. 3.** WidowX robotic arm moving a grabbed object in the bucket on the left.

placed in a given position and released without touching the bucket. The robot must stop also in the case of an unintended collision with other objects or with the robot itself—collisions can be detected using torque estimation from current sensors placed in the joints. Finally, if a general alarm is detected, e.g., by the interaction with a human supervisor, the robot must stop as soon as possible. The manipulator is a 4 degrees-of-freedom Trossen Robotics WidowX arm<sup>8</sup> equipped with a gripper: Fig. 3 shows a snapshot of the robot in the intended usage scenario taken from V-REP<sup>9</sup> simulator. The design of the embedded controller is currently part of the activities related to the “Self-Healing System for Planetary Exploration” use case [16] in the context of the EU project CERBERO.

In this Section, as a basis for our experimental analysis, we consider a set of requirements from the design of an embedded controller for a robotic manipulator. The controller should direct a properly initialized robotic arm—and related vision system—to look for an object placed in a given position and move to such position in order to grab the object; once grabbed, the object is to be moved into a bucket

**Table 2.** Robotic use case requirements synopsis. The table is organized as follows: the first column reports the name of the patterns and it is followed by two groups of three columns denoted with the scope type: the first group refers to the intended specification, the second to the one with fault injections. Each cell in the first group reports the number of requirements grouped by pattern and by scope type. Cells in the second group categorize the 6 injected faults, labeled with F1, . . . , F6.

Pattern	Specification			Fault injections		
	AFTER	AFTER_UNTIL	GLOBALLY	AFTER	AFTER_UNTIL	GLOBALLY
Absence	–	12	14	[F4]	–	[F3]
Existence	9	–	–	–	[F5]	[F4, F6]
Invariant	–	–	29	–	–	[F2, F6]
Precedence	–	–	1	–	–	–
ResponseChain	–	–	2	–	–	–
Response	1	–	4	–	–	[F1]
Universality	2	–	1	–	–	–

<sup>8</sup> <http://www.trossenrobotics.com/widowxrobotarm>.

<sup>9</sup> <http://www.coppeliarobotics.com/>.

In this case study, constrained numerical signals are used to represent requirements related to various parameters, namely angle, speed, acceleration, and torque of the 4 joints, size of the object picked, and force exerted by the end-effector. We consider 75 requirements, including those involving scenario-independent constraints like joints limits, and mutual exclusion among states, as well as specific requirements related to the conditions to be met at each state. The set of requirements involved in our analysis includes 14 Boolean signals and 20 numerical ones. In Table 2 we present a synopsis of the requirements, to give an idea of the kind of patterns used in the specification.<sup>10</sup> While most requirements are expressed with the Invariant pattern, e.g., mutual exclusiveness of states and safety conditions, the expressivity of LTL is required to describe the evolution of the system. Indeed, as shown in [8, 19], it is often the case that few PSPs cover the majority of specifications whereas others are sparsely used.

Our first experiment<sup>11</sup> is to run NUSMV-INVAR on the intended specification translated to LTL( $\mathcal{D}_C$ ). The motivation for presenting the results with NUSMV-INVAR rather than AALTA is twofold: While its performances are worse than AALTA, NUSMV-INVAR is more robust in the sense that it either reaches the time limit or it solves the problem, without ever failing for unspecified reasons like AALTA does at times; second, it turns out that NUSMV-INVAR can deal flawlessly and in reasonable CPU times with all the specifications we consider in this Section, both the intended one and the ones obtained by injecting faults. In particular, on the intended specification, NUSMV-INVAR is able to find a valid model for the specification in 37.1 CPU seconds, meaning that there exists at least a model able to satisfy all the requirements simultaneously. Notice that the translation time from patterns to formulas in LTL( $\mathcal{D}_C$ ) is negligible with respect to the solving time. Our second experiment is to run NUSMV-INVAR on the specification with some faults injected. In particular, we consider six different faults, and we extend the specification in six different ways considering one fault at a time. The patterns related to the faults are summarized in Table 2. In case of faulty specifications, NUSMV-INVAR concludes that there is no model able to satisfy all the requirements simultaneously. In particular, in the case of F2 and F3, NUSMV-INVAR returned the result in 2.1 and 1.7 CPU seconds, respectively. Concerning the other faults, the tools was one order of magnitude slower in returning the satisfiability result. In particular, it spent 16.8, 50.4, 12.2, and 25.6 CPU seconds in the evaluation of the requirements when faults 1, 4, 5 and 6 are injected, respectively.

The noticeable difference in performances when checking for different faults in the specification is mainly due to the fact that F2 and F3 introduce an initial inconsistency, i.e., it would not be possible to initialize the system if they were present in the specification, whereas the remaining faults introduce

<sup>10</sup> The full list of requirements and the fault injection examples are available at <https://github.com/SAGE-Lab/robot-arm-usecase>.

<sup>11</sup> Experiments herein presented ran on a PC equipped with a CPU Intel Core i7-2760QM @ 2.40 GHz (8 cores) and 8 GB of RAM, running Ubuntu 14.04 LTS.

inconsistencies related to interplay among constraints in time, and thus additional search is needed to spot problems. In order to explain this difference, let us first consider fault 2:

*Globally, it is always the case that if `state_init` holds, then not `arm_idle` holds as well.*

It turns out that in the intended specification there is one requirement specifying exactly the opposite, i.e., that when the robot is in `state_init`, then `arm_idle` must hold as well. Thus, the only models that satisfy both requirements are the ones preventing the robot arm to be in `state_init`. However, this is not possible because other requirements related to the state evolution of the system impose that `state_init` will eventually occur and, in particular, that it should be the first one. On the other hand, if we consider fault 6:

*Globally, it is always the case that if `arm_moving` holds, then `joint1_speed > 15.5` holds as well.*  
*Globally, `arm_moving` and `proximity_sensor = 10.0` eventually holds.*

we can see that the first requirement sets a lower speed bound at 15.5 deg/s for `joint1` when the arm is moving, while there exists a requirement in the intended specification setting an upper speed bound at 10 deg/s when the proximity sensor detects an object closer than 20 cm. In this case, the model checker is still able to find a valid model in which `proximity_sensor < 20.0` never happens when `arm_moving` holds, but the second requirements in fault 6 prohibits this opportunity. It is exactly this kind of interplay among different temporal properties which makes NUSMV-INVAR slower in assessing the (in)consistency of some specifications.

## 6 Conclusions

In this paper, we have extended basic PSPs over the constraint system  $\mathcal{D}_C$ , and we have provided an encoding from any  $\text{PSP}(\mathcal{D}_C)$  into a corresponding LTL formula. This enables us to deal with many specifications of practical interest, and to verify them using automated reasoning systems currently available for LTL. Using realistically-sized specifications generated with a probabilistic model we have shown that our approach implemented on the tool AALTA scales to problems containing more than a thousand requirements over hundreds of variables. Considering a real-world case study in the context of the EU project CERBERO, we have shown that it is feasible to check specifications and uncover injected faults, even without resorting to AALTA, but considering NUSMV, a tool which proved to be slower, yet more robust, than AALTA. These results witness that our approach is viable and worth of adoption in the process of requirement engineering. Our next steps toward this goal will include easing the translation from natural language requirements to patterns, and extending the pattern language to deal with other relevant aspects of cyber-physical systems, e.g., real-time constraints. Further elements will also include search for minimum unsatisfiable



cores in requirements, i.e., discovering or approximating the minimum set of requirements causing the inconsistency.

**Acknowledgments.** The research of Luca Pulina and Simone Vuotto has been funded by the EU Commissions H2020 Programme under grant agreement N.732105 (CERBERO project).

## References

1. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). [https://doi.org/10.1007/3-540-45657-0\\_29](https://doi.org/10.1007/3-540-45657-0_29)
2. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. (TOPLAS) **8**(2), 244–263 (1986)
3. Comon, H., Cortier, V.: Flatness is not a weakness. In: Clote, P.G., Schwichtenberg, H. (eds.) CSL 2000. LNCS, vol. 1862, pp. 262–276. Springer, Heidelberg (2000). [https://doi.org/10.1007/3-540-44622-2\\_17](https://doi.org/10.1007/3-540-44622-2_17)
4. Demri, S., DSouza, D.: An automata-theoretic approach to constraint LTL. Inf. Comput. **205**(3), 380–415 (2007)
5. Dillon, L.K., Kutty, G., Moser, L.E., Melliar-Smith, P.M., Ramakrishna, Y.S.: A graphical interval logic for specifying concurrent systems. ACM Trans. Softw. Eng. Methodol. (TOSEM) **3**(2), 131–165 (1994)
6. Dokhanchi, A., Hoxha, B., Fainekos, G.: Metric interval temporal logic specification elicitation and debugging. In: 13th ACM-IEEE International Conference on Formal Methods and Models for Codesign, pp. 21–23 (2015)
7. Dokhanchi, A., Hoxha, B., Fainekos, G.: Formal requirement debugging for testing and verification of cyber-physical systems. arXiv preprint [arXiv:1607.02549](https://arxiv.org/abs/1607.02549) (2016)
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering, pp. 411–420 (1999)
9. Hustadt, U., Konev, B.: TRP++ 2.0: a temporal resolution prover. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 274–278. Springer, Heidelberg (2003). [https://doi.org/10.1007/978-3-540-45085-6\\_21](https://doi.org/10.1007/978-3-540-45085-6_21)
10. Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th International Conference on Software Engineering, pp. 372–381 (2005)
11. Li, J., Pu, G., Zhang, L., Yao, Y., Vardi, M.Y., et al.: Polsat: a portfolio LTL satisfiability solver. arXiv preprint [arXiv:1311.1602](https://arxiv.org/abs/1311.1602) (2013)
12. Li, J., Yao, Y., Pu, G., Zhang, L., He, J.: Aalta: an LTL satisfiability checker over infinite/finite traces. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 731–734 (2014)
13. Li, J., Zhang, L., Pu, G., Vardi, M.Y., He, J.: LTL satisfiability checking revisited. In: 20th International Symposium on Temporal Representation and Reasoning, pp. 91–98 (2013)
14. Li, J., Zhu, S., Pu, G., Vardi, M.Y.: SAT-based explicit LTL reasoning. In: Piterman, N. (ed.) HVC 2015. LNCS, vol. 9434, pp. 209–224. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-26287-1\\_13](https://doi.org/10.1007/978-3-319-26287-1_13)

15. Lumpe, M., Meedeniya, I., Grunske, L.: PSPWizard: machine-assisted definition of temporal logical properties with specification patterns. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 468–471 (2011)
16. Masin, M., Palumbo, F., Myrhaug, H., de Oliveira Filho, J., Pastena, M., Pelcat, M., Raffo, L., Regazzoni, F., Sanchez, A., Toffetti, A., et al.: Cross-layer design of reconfigurable cyber-physical systems. In: 2017 Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 740–745. IEEE (2017)
17. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE (1977)
18. Pnueli, A., Manna, Z.: The temporal logic of reactive and concurrent systems. Springer **16**, 12 (1992)
19. Post, A., Hoenicke, J.: Formalization and analysis of real-time requirements: a feasibility study at BOSCH. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 225–240. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-27705-4\\_18](https://doi.org/10.1007/978-3-642-27705-4_18)
20. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 149–167. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-73370-6\\_11](https://doi.org/10.1007/978-3-540-73370-6_11)
21. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. *Int. J. Softw. Tools Technol. Transf. (STTT)* **12**(2), 123–137 (2010)
22. Rozier, K.Y., Vardi, M.Y.: A multi-encoding approach for LTL symbolic satisfiability checking. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 417–431. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-21437-0\\_31](https://doi.org/10.1007/978-3-642-21437-0_31)
23. Schwendimann, S.: A new one-pass tableau calculus for **PLTL**. In: de Swart, H. (ed.) TABLEAUX 1998. LNCS (LNAI), vol. 1397, pp. 277–291. Springer, Heidelberg (1998). [https://doi.org/10.1007/3-540-69778-0\\_28](https://doi.org/10.1007/3-540-69778-0_28)



# Automatic Generation of DO-178 Test Procedures

César Ochoa Escudero<sup>1,2(✉)</sup>, Rémi Delmas<sup>1</sup>, Thomas Bochot<sup>2</sup>,  
Matthieu David<sup>2</sup>, and Virginie Wiels<sup>1</sup>

<sup>1</sup> ONERA, Toulouse, France

{cesar.ochoa,remi.delmas,virginie.wiels}@onera.fr

<sup>2</sup> Liebherr-Aerospace Toulouse SAS, Toulouse, France

{cesar.ochoaescudero,thomas.bochot,matthieu.david}@liebherr.com

**Abstract.** The work presented in this paper takes place in the context of the testing activities of safety critical Air Management Systems for civilian and military aircraft. The applicative software of such systems is developed following DO-178 guidelines, using a model-based approach built on the SCADE modeling language. In the current V&V process, *Test Cases* (TCs) specify test conditions and expected outcomes on internal data-flows of the SCADE model. TCs are then implemented in the form of concrete *Test Procedures* (TPs) that are run against the executable object code and can thus only drive the main inputs of the program. TP implementation is a complex task, today performed manually. This paper proposes an approach to assist the generation of TPs, based on a purpose-built domain specific language for test case specification, from which synchronous observers are generated and composed with the applicative software SCADE model. TPs are then obtained by using a model checker to refute the observer output, yielding, after some post-processing a trace of main input values extended with expected outcome checks.

**Keywords:** Software testing · Domain-specific language  
Synchronous observers · Model-checking

## 1 Introduction

Liebherr-Aerospace Toulouse (LTS) is a provider of *Air Management Systems* (AMS) for civilian and military aircraft. The main functions of an AMS are to provide air supply, wing anti-icing, environmental control and cabin pressurization for the crew and passengers. Consequently, an AMS implements several sub-functions such as data acquisition, consolidation, control, monitoring, etc. The safety-critical aspect of the AMS requires its software development process to comply with DO-178 DAL-B guidelines. To this end, the Liebherr software testing process, depicted in Fig. 1, is built on the following notions:

- (1) *High-Level Requirements* (HLRs) are the requirements of the AMS, which must be implemented and tested, and are written in natural language;

- (2) *Low-Level Requirements* (LLRs) are detailed software specifications. In the current LTS process, most LLRs are formalized as SCADE [1] data-flow models. All LLRs together form what we call the *SCADE model* of the applicative software. An HLR can be refined in more than one LLR, and a given LLR can support more than one HLR. The Executable Object Code is automatically generated from the SCADE model using suitable code generator and compiler;
- (3) *Test Cases* (TCs) are declarative *test specifications*, written in natural language. TCs are linked with HLRs. Any HLR must be covered by one or more TCs. The test conditions and expected outcomes are expressed in terms of HLR identifiers. Therefore its writing is independent from the SCADE LLRs.
- (4) *Test Procedures* (TPs) are *executable implementations* of TCs, and are meant to be run against the executable applicative software, on the target computer. A TP implements several TCs in sequence, drives the software main inputs and checks the outcomes specified in the TCs.

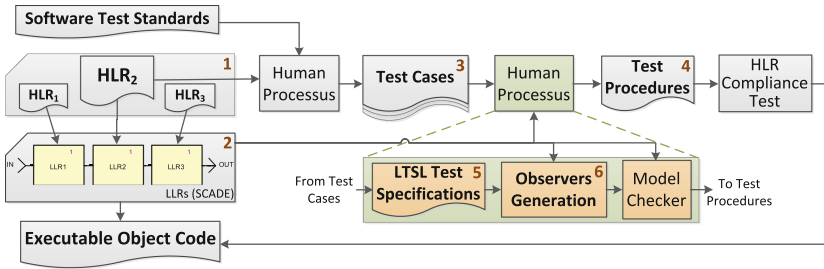


Fig. 1. Software testing process

The difficulty of implementing TPs is mainly due to the fact that TPs are restricted to drive only the software main inputs, which in turn makes TPs highly dependent on the overall SCADE data-flow structure, even-though the TCs to be implemented are specified independently of the SCADE data-flow structure. Indeed, in order to establish the required test conditions on internal data-flows, one must perform a sort of back-propagation reasoning through the overall data-flow structure to derive appropriate main input values for the TP. This task is mainly performed by human operators today, and is hence very resource consuming. Furthermore this data-flow structure dependency makes TPs *fragile*: a TP for a given HLR might be invalidated by evolutions of seemingly indirectly related HLRs/LLRs, or by non-functional evolution of the model, for instance by subsampling of its sub-functions. To illustrate fragility, let us come back to Fig. 1 and focus on  $HLR_2$ , that is refined into  $LLR_2$ . A modification of  $HLR_1/LLR_1$  might invalidate the TPs corresponding to  $HLR_2/LLR_2$ , since  $LLR_1$  lies in the cone of influence of  $LLR_2$ .

The work presented in this paper aims at reducing the human effort involved in producing and maintaining TPs under evolutions of corresponding HLRs/LLRs. First, we propose a formal and declarative language, named *Liebherr Test Specification Language* (LTSL) – item (5) in Fig. 1, used as an intermediate formalization step between natural language TC specification and TP implementation. Second, we provide an automatic generator of synchronous SCADE observers from LTSL specifications, which are then used to instrument the SCADE model – item (6) in Fig. 1. A model-checker (Systemel S3 [2]) is then used on the instrumented model to refute the observer output, yielding traces for the software main inputs which satisfy the test conditions expressed in the LTSL specification. These traces are then post-processed to be augmented with checks of expected test outcomes to obtain TPs implementations.

The paper is structured as follows: Sect. 2 provides information on the industrial context in which this work takes place. Section 3 defines the LTSL syntax and semantics. The automatic generation of SCADE observers from LTSL specifications is discussed in Sect. 4. Section 5 provides details on the implementation of the approach and illustration. Section 6 discusses related works. Last, Sect. 7 concludes the paper and outlines some perspectives to this work.

## 2 Industrial Context

In this section, we first provide an example of a high-level requirement used throughout the paper for illustration, and then present a more detailed look into the software verification process currently in use at Liebherr-Aerospace Toulouse.

### 2.1 Running HLR Example

The Bleed system is a sub-system of the AMS. Its purpose is to bleed hot air from compressor stages of the engines and to condition it for later use as input for the wing anti-icing sub-system or the input cabin pressurization system. Table 1 presents the HLR for the function *Monitoring of Bleed Overpressure*, a prototypical example of a latched monitoring function (largely simplified with respect to the industrial version by lack of space).

Req1 states that the function is required to run at a 500 ms period; to generate an alarm when A is set for at least 500 ms; to be reset when B is set and A is not set, or when C is not set, with no confirmation time; priority is given to the reset conditions.

### 2.2 Software Test Standards

The goal of the software testing process is to demonstrate that the Executable Object Code satisfies its HLRs. The *Software Test Standards* (STS) is an internal LTS document containing operational procedure which is deemed sufficient for DO-178 compliance. It defines rules to apply when redacting TCs and implementing TPs for HLRs. The following STS rules are relevant to our work:

**Table 1.** Monitoring of Bleed Overpressure HLR

Req1	
OVPR is set every 500 ms according to the following logic:	
▷ Set logic:	A is set
▷ Set confirmation time:	500 ms
▷ Reset logic:	○ B is set and A is unset ○ C is unset
▷ Reset confirmation time:	None
▷ Priority:	Reset

1. TCs shall be defined only with reference to HLRs and HLR variable identifiers; 2. The *purpose* of each TC, which details which aspect of an HLR is addressed by the TC, shall be documented; 3. No two TCs shall have the same purpose; 4. The values of the controlled inputs and the expected test outcomes must be explicitly stated in the TC; 5. TPs shall be defined with reference to LLR interface identifiers; 6. Several TCs can be implemented in sequence within a single TP, and TCs shall be uniquely identified in the context of a TP; 7. In TPs, the (sequence of) expected outcome(s) of two consecutive TCs must be different;

The STS document also provides recommendations regarding structural coverage: 1. Each TC should test the effect of each input on the output value of tested function independently; 2. For combinational logics with boolean inputs, a TC should be specified for each possible entry of logics truth table; 3. For threshold logics (*i.e.* range checking logic, confirmation logic, *etc.*), the logics output value shall be checked with input(s) below the threshold value(s), as well as with input(s) above the threshold value(s).

### 2.3 Running Example: TCs for Req1

Following the current process, five TCs are written to cover HLR Req1 and then grouped and sequenced in Table 2. The first row specifies the identifier of the HLR covered by the TCs. The second row specifies a sequence of unique TC identifiers *TC1* to *TC5*. The third row documents the purpose of each TC. Each TC consists of a sequence of one or more *Trace Snippets*; (TSS). A TS is characterized by minimum duration expressed as a number of HLR periods (fourth row), a vector of input values to be maintained over that duration (fifth row), and a collection of checks to be performed at some specific instant relative to the start of the TS (which has index 0).

The TCs grouped in a table are implemented in a single TP. Consequently the order of TCs and TSs is significant, each TC relying on the state reached through previous TCs to achieve its purpose. For instance, *TC3* tests a *maintain* behavior, using a first TS to test the initial setting of the latch, and a second TS

**Table 2.** TCs covering Req1 as per Software Test Standards

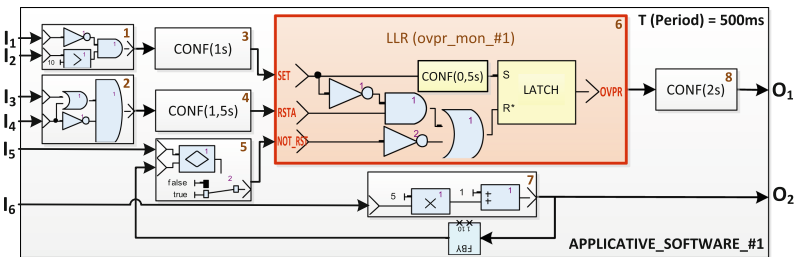
HLR		Req1							
TC id		TC1		TC2	TC3		TC4	TC5	
Purpose		Set logic		1 <sup>st</sup> rst log	Maintain set logic		2 <sup>nd</sup> rst log	Reset priority	
Duration at least		1 cycle	2 cycles	1 cycle	2 cycles	1 cycle	1 cycle	2 cycles	1 cycle
Inputs	A	False	True	False	True	False	False	True	True
	B	False	True	True	True	False	False	True	True
	C	False	True	True	True	True	False	True	False
E.Outp.	OVPR	False at cycle 0	False at cycle 0 True at cycle 1	False at cycle 0	False at cycle 0 True at cycle 1	True at cycle 0	False at cycle 0	False at cycle 0 True at cycle 1	False at cycle 0

to test latching after set conditions are released. Hence *TC3* must be started in a state in which the latch is not already set, which corresponds to the expected outcome of *TC2*. Specifying the order of TCs is the responsibility of the test engineer.

**2.4 Running Example: TP Implementation for Req1**

This sub-section illustrates back-propagation reasoning required to implement TPs, as performed currently by human operators. Figure 2 shows the Applicative Software SCADE model, consisting of LLRs  $LLR_1$  to  $LLR_8$  (SCADE nodes). The LLR of interest here is  $LLR_6$ , with SCADE node instance path `APPLICATIVE_SOFTWARE_#1/ovpr_mon_#1`, and implements HLR Req1.

First, the naming correspondence between HLR and LLR is established based on traceability documents, as follows:  $A \rightarrow SET$ ,  $B \rightarrow RSTA$ ,  $C \rightarrow NOT\_RST$ ,  $OVPR \rightarrow OVPR$ . Since the TP can only drive main inputs  $I = \{I_i | i \in [1, 6]\}$   $I_1$ , the human implementor must analyze the cone of influence (COI) of `SET`, `RSTA` and `NOT_RSTA` to determine how to drive them from the main inputs. The COI might be arbitrarily complex, involving boolean logic, arithmetic operations, timing, hysteresis, latching, *etc.* Here,  $COI(SET) \cap I = \{I_1, I_2\}$ ,  $COI(RSTA) \cap I = \{I_3, I_4\}$ , and  $COI(NOT\_RST) \cap I = \{I_5, I_6\}$ . Let us now focus on implementing *TC1* on  $LLR_6$ . It's decomposed in two consecutive TSs. The first one is an



**Fig. 2.** Applicative software SCADE model

initialization logic specifying that inputs SET, RSTA and NOT\_RSTA must be false simultaneously for at least 1 cycle. Since the first ones are outputs of  $LLR_3$  and  $LLR_4$  they are false by default. Leaving the case of NOT\_RSTA, which requires  $I_5 \neq fby(I_6 * 5 + 1, 1, 10)$ , i.e.  $I_5 \neq 10$  when in the initial state. The following TS of *TC1* specifies that inputs SET, RSTA and NOT\_RSTA must be true simultaneously for at least 2 cycles. Consequently the input of  $LLR_3$  must be true for at least  $2 + 2 = 4$  cycles, which requires  $\neg I_1 \wedge I_2 > 10$  to hold for at least 4 cycles – let us fix  $I_1 = false$  and  $I_2 = 20$ ; Consequently the input of  $LLR_4$  must be set to true for at least  $3 + 2 = 5$  cycles, which requires  $(I_3 \vee I_4) \wedge \neg I_4$  to hold for 5 cycles – let us fix  $I_3 = false$ ,  $I_4 = true$ ; Consequently the inputs of  $LLR_5$  must satisfy  $I_5 \neq fby(I_6 * 5 + 1, 10, 1)$  for at least 2 cycles – let us fix  $I_5 = 50$  and  $I_6 = 10$ . Last, to establish all these conditions simultaneously for 2 cycles, we need to maintain the appropriate main input values for  $max(2, 4, 5) = 5$  cycles. So even though *TC1*'s specified duration is 2 cycles, the extra duration of 3 cycles is a function of the COI of inputs controlled in the test case, which cannot be determined when specifying the TCs, but only when implementing the TP. The process is repeated for *TC2* to *TC5* in order to obtain the full TP for Req1. In general, such extra duration can be required to transition between any two consecutive trace snippet conditions.

**Table 3.** Fragment of TP implementation for Req1 in TCL language

```

;# Scenario for TC1
SSM::set APPLICATIVE_SOFTWARE_#1/I1 false
SSM::set APPLICATIVE_SOFTWARE_#1/I2 20
SSM::set APPLICATIVE_SOFTWARE_#1/I3 false
SSM::set APPLICATIVE_SOFTWARE_#1/I4 true
SSM::set APPLICATIVE_SOFTWARE_#1/I5 50
SSM::set APPLICATIVE_SOFTWARE_#1/I6 10
Cycle 1
;# Check Inputs:
Check "APPLICATIVE_SOFTWARE_#1/ovpr_mon_#1/SET" false
Check "APPLICATIVE_SOFTWARE_#1/ovpr_mon_#1/RSTA" false
Check "APPLICATIVE_SOFTWARE_#1/ovpr_mon_#1/NOT_RST" false
;# Check Outputs:
Check "APPLICATIVE_SOFTWARE_#1/ovpr_mon_#1/OVPR" false
Cycle 3

;# Check Inputs:
Check "APPLICATIVE_SOFTWARE_#1/ovpr_mon_#1/SET" true
Check "APPLICATIVE_SOFTWARE_#1/ovpr_mon_#1/RSTA" true
Check "APPLICATIVE_SOFTWARE_#1/ovpr_mon_#1/NOT_RST" true
;# Check Outputs:
Check "APPLICATIVE_SOFTWARE_#1/ovpr_mon_#1/OVPR" false
Cycle 1
;# Check Inputs:
Check "APPLICATIVE_SOFTWARE_#1/ovpr_mon_#1/SET" true
Check "APPLICATIVE_SOFTWARE_#1/ovpr_mon_#1/RSTA" true
Check "APPLICATIVE_SOFTWARE_#1/ovpr_mon_#1/NOT_RST" true
;# Check Outputs:
Check "APPLICATIVE_SOFTWARE_#1/ovpr_mon_#1/OVPR" true
;# Scenario for TC2
...

```

In practice, a TP is implemented as sequential program in the TCL scripting language as shown in Table 3. The certification process requires, in addition to checking the expected test outcomes, to check that the LLR inputs have been controlled as expected from the main inputs.

### 3 Liebherr Test Specification Language

This section details the syntax and semantics of the *Liebherr Test Specification Language* (LTSL) Domain Specific Language, meant as an intermediary formalization step between TC specification and TP implementation.



### 3.1 Syntax

Since LTSL is an intermediary step between TCs and TPs, it inherits some of the STS requirements specified in Sect. 2.2: 1. An LTSL file virtually defines the scope of a single TP so the TCs shall be uniquely identified within an LTSL file; 2. The *purpose* of each TC must be documented in the LTSL format; 3. The values of the controlled inputs and the expected test outcomes for each TC must be explicitly stated in the LTSL format; LTSL is a step closer to a TP implementation so it uses LLR identifiers and not HLR, however both HLR and LLR are declared at the top of an LTSL file for traceability.

The LTSL syntax is defined below using an EBNF notation where: 1. parentheses  $()$  represent grouping; 2. pipes  $|$  represent disjunction; 3. square braces  $[ r ]$  represent 0 or 1 repetitions; 4. curly braces  $\{ r \}$  represent 0 or more repetitions.

First identifiers, value literals and string literals are defined:

$$\begin{aligned} \langle ID \rangle &::= \text{regexp: } [_a-zA-Z]^+ & \langle INTP \rangle &::= \text{regexp: } [1-9] [1-9]^* \\ \langle INT \rangle &::= \text{regexp: } -?[0-9]^+ & \langle REAL \rangle &::= \text{regexp: } -?[0-9]^+ \backslash . [0-9]^+ \\ \langle STR \rangle &::= \text{regexp: } \" [^\"]^* \"/>$$

An LTSL file starts with an *HLR* identifier and corresponding SCADe instance graph identifier of the *LLR*. Then it specifies a sequence of *TC* sections, each described by an identifier, a purpose string and one or more trace snippets. A trace snippet is characterized by a *duration*, an *observe block* and a *check block*:

$$\begin{aligned} \langle ltslFile \rangle &::= \text{HLR : } \langle ID \rangle ; \text{LLR : } \langle llrID \rangle ; \{ \langle testCase \rangle \} \\ \langle llrID \rangle &::= \langle ID \rangle \_ \# \langle INTP \rangle \{ / \langle ID \rangle \_ \# \langle INTP \rangle \} \\ \langle testCase \rangle &::= \text{Test case } \langle ID \rangle : \langle STR \rangle \{ \langle traceSnippet \rangle \}^+ \\ \langle traceSnippet \rangle &::= \text{For } \langle duration \rangle : \langle obsBlock \rangle [ \langle chekBlock \rangle ] \end{aligned}$$

The *duration* of a trace snippet is an *interval* where bounds are expressed in number of ticks of the LLR clock, with the lower bound strictly positive. The upper bound is optional – absence is represented by the “*don’t care*” symbol  $?$ . This is meant to allow for unknown latencies to occur when transitioning from one trace snippet to the next.

$$\langle duration \rangle ::= \langle interval \rangle \text{ cycles} \quad \langle interval \rangle ::= [ \langle INTP \rangle , ( \langle INTP \rangle | ? ) ]$$

An *observe block* lists conditions that must hold over LLR input variables for the duration of the snippet, within some numerical tolerance for real-valued variables:

$$\langle obsBlock \rangle ::= \text{Observe : } \{ \langle cond \rangle ; \} \quad \langle cond \rangle ::= \langle ID \rangle = \langle val \rangle [ \tau = \langle REAL \rangle ]$$

A *check block* specifies a list of checks to be performed over a LLR output variables at some position of the trace snippet. The position represents a number

of execution cycles relative to the beginning of the snippet (first cycle has index 0), and must be strictly less than the lower bound:

$$\langle \text{chkBlock} \rangle ::= \text{Check} : \{ \langle ID \rangle; \langle \text{cond} \rangle \langle \text{rlPos} \rangle; \} \quad \langle \text{rlPos} \rangle ::= \text{at cycle} \langle INT \rangle$$

### 3.2 Semantics

We now define a formal *trace semantics* for LTSL, that is, the formal conditions according to which a concrete program execution trace *satisfies* an LTSL specification. Let  $V$  be the set of variables, partitioned in  $V = V_{\mathbb{B}} \cup V_{\mathbb{Z}} \cup V_{\mathbb{R}} \cup \text{ChkLbl}$ , with  $V_{\mathbb{B}}$  a set of boolean-valued variables,  $V_{\mathbb{Z}}$  a set of integer-valued variables,  $V_{\mathbb{R}}$  a set of real-valued variables,  $\text{ChkLbl}$  a set of *check labels*. Let  $S$  be a set of states, equipped with evaluation functions  $eval_{\mathbb{B}} : S \times V_{\mathbb{B}} \rightarrow \mathbb{B}$ ,  $eval_{\mathbb{Z}} : S \times V_{\mathbb{Z}} \rightarrow \mathbb{Z}$ ,  $eval_{\mathbb{R}} : S \times V_{\mathbb{R}} \rightarrow \mathbb{R}$  and  $eval_{\text{ChkLbl}} : S \times \text{ChkLbl} \rightarrow \mathbb{B}$  which allow to obtain the value of a variable in a given state. Let  $Traces = \{ tr = [s_0, \dots, s_n] \mid s_i \in S \}$  be the set of *traces*, *i.e.* of all possible finite sequences of states. Let  $Conds$  be the set of *conditions*, equipped with functions  $var : Conds \rightarrow V_{\mathbb{B}} \cup V_{\mathbb{Z}} \cup V_{\mathbb{R}}$ ,  $val : Conds \rightarrow \mathbb{B} \cup \mathbb{Z} \cup \mathbb{R}$  and  $tol : Conds \rightarrow \mathbb{Z} \cup \mathbb{R}$  which return respectively the variable, the value and the tolerance of the condition. Let  $\text{ChkCnds}$  be the set of *check conditions* equipped with functions  $label : \text{ChkCnds} \rightarrow \text{ChkLbl}$ ,  $ccnd : \text{ChkCnds} \rightarrow Conds$ , and  $relpos : \text{ChkCnds} \rightarrow \mathbb{N}$ , which return respectively the label, the condition and position of the check condition. Let  $\text{TraceSnippets}$  be the set of all possible *trace snippets*, equipped with functions  $lwr : \text{TraceSnippets} \rightarrow \mathbb{N}^+$ ,  $upr : \text{TraceSnippets} \rightarrow \mathbb{N}^+ \cup \{?\}$ ,  $oCnds : \text{TraceSnippets} \rightarrow Conds^*$  and  $cCnds : \text{TraceSnippets} \rightarrow \text{ChkCnds}^*$  which return respectively the lower duration bound, optional upper bound, the set of observed conditions and the set of checked conditions of a trace snippet. Let  $Spec$  be the set of all possible *specifications* where a *specification* is a finite sequence of TCs, each of which in turn is a finite sequence of TSs. The function  $trSnpts : Spec \rightarrow \text{TraceSnippets}^*$  be the function that returns the concatenation of all TSs contained in a specification.

**Definition 1 (Condition Evaluation Function).** *The following function allows evaluating a condition in a given state:  $cEval : S \times Conds \rightarrow \mathbb{B}$*

$$cEval(s, c) = \begin{cases} eval_{\mathbb{B}}(s, var(c)) = val(c) & var(c) \in V_{\mathbb{B}} \\ eval_{\mathbb{Z}}(s, var(c)) = val(c) & var(c) \in V_{\mathbb{Z}} \\ abs(eval_{\mathbb{R}}(s, var(c)) - val(c)) \leq tol(c) & var(c) \in V_{\mathbb{R}} \end{cases}$$

**Definition 2 (LTSL Satisfaction Relation).**

Let  $tr \in Traces$ ,  $spec \in Spec$  and  $trSnpts(spec) = [ \overbrace{ts_1}^{\text{head}}, \overbrace{ts_2, \dots, ts_n}^{\text{tail}} ]$  be the sequence of trace snippets to be evaluated against  $tr$  for satisfaction.

Let  $i$  be an integer representing an index in the finite trace  $tr$ . Then  $(tr, i) \models spec$  if and only if  $\exists k, k \geq lwr(\text{head})$  and  $k \leq upr(\text{head})$  if  $upr(\text{head})$  is defined, such that:

- $length(tr) - i \geq k$  and
- $\forall obsCond \in oConds(head), \forall j \in [i, i + k - 1], cEval(s_j, obsCond) = \top$  and
- $\forall chkCond \in cConds(head)$ :
  - let  $j = relpos(chkCond)$
  - $\forall s \neq s_{i+j-1}, eval_{ChkLbl}(s, clabel(chkCond)) = \perp$  and
  - $eval_{ChkLbl}(s_{i+j-1}, clabel(chkCond)) = \top$  and
  - $cEval(s_{i+j-1}, ccnd(chkCond)) = \top$
- If  $tail \neq \emptyset$ ,  $(spec, i + k) \models tail$ , i.e. the trace suffix starting at index  $i + k$  satisfies the tail of the specification if it is not empty.
- Lastly, we define:  $tr \models spec \equiv \exists start\_index \in \mathbb{N}, (tr, start\_index) \models spec$ , i.e. a trace satisfies a specification if and only if the trace satisfies the specification starting some finite index  $start\_index$ .

The existentially quantified variable  $k$  associated with each TS in the above definition is called the **span variable** of the TS.

### 3.3 Running Example: Req1 TCs Expressed in LTSL

Figure 3 presents the LTSL formalization of the Req1 TCs. The duration of each trace snippet has a “don’t care” upper bound to allow maximum flexibility in the specified trace. One could alternatively have forced strict upper bounds on trace snippets, in effect reducing the set of admitted traces.

<pre> HLR:Req1; LLR:APPLICATIVE_SOFTWARE_#/ovpr_mon_#1; Test case TC1: "set logic" For [1,?] cycles: Observe:   SET=false; RSTA=false; NOT_RST=false; Check:   Chk0: OVPR = false at cycle 0; For [2,?] cycles: Observe:   SET=true; RSTA=true; NOT_RST=true; Check:   Chk1: OVPR = false at cycle 0;   Chk2: OVPR = true at cycle 1; Test case TC2: "first reset logic" For [1,?] cycles: Observe:   SET=false; RSTA=true; NOT_RST=true; Check:   Chk3: OVPR = false at cycle 0; Test case TC3: "maintain set logic" For [2,?] cycles: Observe:   SET=true; RSTA=true; NOT_RST=true; Check:   Chk4: OVPR = false at cycle 0; </pre>	<pre> Chk5: OVPR = true at cycle 1; For [1,?] cycles: Observe:   SET=false; RSTA=false; NOT_RST=true; Check:   Chk6: OVPR = true at cycle 0; Test case TC4: "second reset logic" For [1,?] cycles: Observe:   SET=false; RSTA=false; NOT_RST=false; Check:   Chk7: OVPR = false at cycle 0; Test case TC5: "reset priority" For [2,?] cycles: Observe:   SET=true; RSTA=true; NOT_RST=true; Check:   Chk8: OVPR = false at cycle 0;   Chk9: OVPR = true at cycle 1; For [1,?] cycles: Observe:   SET=true; RSTA=true; NOT_RST=false; Check:   Chk10: OVPR = false at cycle 0; </pre>
--	--

Fig. 3. LTSL specification of Req1 TCs.

## 4 SCADE Observers Generation

This section details the translation rules used to generate a synchronous SCADE observer from an LTSL specification.

## 4.1 SCADE Automata Syntax

The SCADE language is a formal graphical notation based on Lustre [3, 12]. As of version 6, the SCADE language offers synchronous hierarchical state machines [4]. Our translation requires only a subset of the full SCADE 6 language, defined below. First, we define some recurrent patterns, such as built-in type identifiers, numeral values and variable declarations:

$$\begin{aligned}
 \langle ID \rangle &::= \mathbf{regexp}: [_a-zA-Z0-9]^+ & \langle T \rangle &::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{real} \\
 \langle REAL \rangle &::= \mathbf{regexp}: -?[0-9]^+\backslash.[0-9]^+ & \langle INT \rangle &::= \mathbf{regexp}: -?[0-9]^+ \\
 \langle val \rangle &::= \mathbf{true} \mid \mathbf{false} \mid \langle INT \rangle \mid \langle REAL \rangle & \langle vD \rangle &::= \langle ID \rangle : \langle T \rangle \\
 \langle dftD \rangle &::= \langle vD \rangle [ \mathbf{default} = \langle val \rangle ]
 \end{aligned}$$

A SCADE program is a collection of *type*, *constant* and *operator* declarations.

$$\begin{aligned}
 \langle opD \rangle &::= \mathbf{node} \langle ID \rangle \langle inD \rangle \mathbf{returns} \langle outD \rangle \langle locD \rangle \langle body \rangle \\
 \langle typeD \rangle &::= \mathbf{type} \langle ID \rangle = \langle T \rangle; & \langle constD \rangle &::= \mathbf{const} \langle vD \rangle = \langle val \rangle; \\
 \langle inD \rangle &::= \langle \langle vD \rangle \{ ; \langle vD \rangle \} \rangle & \langle locD \rangle &::= \mathbf{var} \{ \langle dftD \rangle ; \} \\
 \langle outD \rangle &::= \langle \langle dftD \rangle \{ ; \langle dftD \rangle \} \rangle & \langle body \rangle &::= \mathbf{let} \{ \langle dfD \rangle \} \mathbf{tel}
 \end{aligned}$$

The body of an operator contains a collection of *data-flow definitions*, which can be either a pure data-flow definition or an automaton definition. The subset of data-flow *expressions* we use is limited to constants, scalar variables, operator instantiation, logical connectors, initialization, unit delay operators and relational operators.

$$\begin{aligned}
 \langle dfD \rangle &::= \langle ID \rangle = \langle e \rangle ; \mid \langle autoD \rangle \\
 \langle e \rangle &::= \langle ID \rangle \mid \langle val \rangle \mid \mathbf{pre}(\langle e \rangle) \mid \langle ID \rangle(\langle e \rangle) \mid \mathbf{not}(\langle e \rangle) \mid \langle e \rangle (\mathbf{and} \mid \mathbf{or} \mid \mathbf{\rightarrow} \mid \mathbf{|\leq}) \langle e \rangle
 \end{aligned}$$

An *automaton* contains a collection of *states*, one of which must be tagged *initial*. A state has a list of outgoing transitions in which order reflects priority, and a body consisting of a collection of scalar data-flow definitions. A transition is characterized by its guard expression and target state. We only use *strong unless transitions*, for which guard evaluation and firing occur in the same logical step. Strong transitions require avoiding causality loops when using flows defined by the automaton itself in transition guards, by introducing unit delays in guards when required.

$$\begin{aligned}
 \langle autoD \rangle &::= \mathbf{automaton} \{ \langle state \rangle \} \mathbf{returns} \dots; \quad \langle trs \rangle ::= \mathbf{if} \langle e \rangle \mathbf{resume} \langle ID \rangle \\
 \langle state \rangle &::= [ \mathbf{initial} ] \mathbf{state} \langle ID \rangle \mathbf{unless} \{ \langle trs \rangle \} \mathbf{let} \{ \langle ID \rangle = \langle e \rangle \} \mathbf{tel}
 \end{aligned}$$

## 4.2 Translation Function

In this section, we define the function translating an LTSL specification to a SCADE observer.

We use the notation  $\text{prtSet}(\{e_1, \dots, e_n\}, \text{sep}) \equiv e_1 \text{ sep } \dots \text{ sep } e_n$  for the concatenation of SCADE expressions  $e_i$  with separator  $\text{sep}$ ; as well as  $\text{and}_{x \in S} e[x] \equiv e[x_1] \text{ and } \dots \text{ and } e[x_n]$  for the n-ary conjunction of SCADE expression  $e$  dependent on  $x$  ranging over some set  $S$ .

Reusing Sect. 3.2 notations, let  $\text{trSnpts}(\text{spec}) = [ts_1, \dots, ts_n]$ , where  $\text{Spec} \in \text{Spec}$  is an LTLS *specification*. The  $\text{genSCADEObservers}$  translation function generates an observer node from  $\text{spec}$ :

```

genSCADEObservers(spec)  $\triangleq$ 
node observer(prtSet(inD(spec)  $\cup$  {trig : bool}, ;))
  returns(Incomplete : bool default = true;
    prtSet({clabel(chk) : bool default = false | chk  $\in$  cConds(tsi)}, ;))
var cpt : int default = 0;
let
  automaton
    initial state S0 unless if obCnd(ts1) resume S1;
    prtSet({state(tsi, tsi+1) | i  $\in$  [1, n - 1]}, \n)
    state Sn unless
      if abRg(tsn) or (not(obCnd(tsn)) and not(inRg(tsn))) resume S0;
      if inRg(tsn) and trign resume Final;
      let ctDf(tsn); prtSet({lblDf(c) | c  $\in$  cConds(tsn)}, ;) tel
    state Final let Incomplete = false; tel
  returns ..;
tel

```

The observer input flows are generated by the function:

$$\text{inD}(\text{spec}) \triangleq \{\text{var}(c) : \text{otp}(c) \mid c \in \text{oConds}(ts), ts \in \text{trSnpts}(\text{spec})\}$$

which gathers all LLR identifiers occurring in the LTSL specification, their type being inferred using function  $\text{otp} : \text{Conds} \rightarrow \{\text{bool} \mid \text{int} \mid \text{real}\}$ . The trigger variable is an extra free observer input, whose utility is detailed later in the *state* function definition.

The observer defines a boolean output flow per check label occurring in the LTSL specification, as well as a boolean output flow **Incomplete**, which is defined to be true by default and false when a trace satisfying the LTSL specification was observed and the Final state was reached.

The observer contains an integer counter **cpt** defined in each state to count the number of cycles spent in that state. Our encoding uses two auxiliary nodes for checking an input  $i$  against a range  $[\text{LB} - 1, \text{UB} - 1]$ , where **UB** is possibly a “don’t care” value, represented as  $\text{UB} = -1$  by the encoding. Subtracting 1 from each bound is a consequence of using these checks in strong transitions, and is used to compensate for checking the previous value of the local counter

– see transition code below. The node `InRange` checks if `i` is in the range, or simply above the lower bound when the upper bound is a “don’t care”. The node `AbRange` checks if the input `i` has gone above the upper bound of the range, which is false when the upper bound is a “don’t care”.

```
node InRange (i : int; LB : int; UB : int) returns (IR : bool)
let IR= i >= LB - 1 and (if UB = -1 then true else i <= UB - 1); tel
node AbRange (i : int; UB : int) returns (AR : bool)
let AR= if UB = -1 then false else i > UB - 1; tel
```

Based on the precedent library, the following intermediary functions are defined. Functions  $inRg(ts_i)$  (resp.  $abRg(ts_i)$ ) generates an expression which checks if the counter value corresponds to an allowed number of execution cycles for  $ts_i$  (resp. if the counter value exceeds the number of execution cycles allowed for the  $ts_i$ ). These expressions are later used in strong transition guards and require to introduce a unit delay on `cpt` to avoid causality loops (since `cpt` is defined in the state itself). The  $cSat$  function generates a saturation value for the counter, equal to the upper bound of  $ts_i$  duration when it’s defined or to its lower bound otherwise. One is added to that value to allow detecting overstaying in a state. The  $obCnd$  expression checks if the observed conditions corresponding to  $ts_i$  are satisfied.

$$inRg(ts_i) \triangleq \text{InRange}((-1) \rightarrow (\text{pre } \text{cpt}), lwr(ts_i), upr(ts_i))$$

$$abRg(ts_i) \triangleq \text{AbRange}((-1) \rightarrow (\text{pre } \text{cpt}), upr(ts_i))$$

$$cSat(ts_i) \triangleq \text{if } upr(ts_i) = -1 \text{ then } lwr(ts_i)+1, \text{ else } upr(ts_i)+1$$

$$obCnd(ts_i) \triangleq \text{and}_{c \in oConds(ts_i)} var(c) = val(c)$$

The body of the observer consists of a single state machine. The purpose of the initial state  $S_0$  is to wait until the TS  $ts_1$  observed conditions are satisfied, and to transition to state  $S_1$ . The definitions of states  $S_1$  to  $S_{n-1}$  are obtained with the function *state* which generates a state  $S_i$  encoding TS  $ts_i$ :

```
state(ts_i, ts_{i+1}) \triangleq
state Si unless
  if abRg(ts_i) or ((not obCnd(ts_i)) and (not obCnd(ts_{i+1})))
    or ((not obCnd(ts_i)) and (not inRg(ts_i))) resume S0;
  if inRg(ts_i) and obCnd(ts_{i+1}) and trig resume Si+1;
let
cpt=0 \rightarrow (if (pre cpt + 1 <= cSat(ts_i)) then (pre cpt + 1) else cSat(ts_i));
prtSet({label(chk) = cpt = relpos(chk); | chk \in cConds(ts_i)}, \n)
tel
```

Two transitions allow to exit  $S_i$ . The priority is given to the *error transition* back towards the initial state  $S_0$ . It is fired in the following cases: 1. The number of

execution cycles spent in this state is greater than the allowed number of cycles for  $ts_i$ ; 2. The observed conditions corresponding to  $ts_i$  and  $ts_{i+1}$  are both not satisfied; 3. The number of execution cycles spent in this state are lesser than the allowed number of cycles for  $ts_i$  and the observed conditions corresponding to  $ts_{i+1}$  are not satisfied. The second transition to  $S_{i+1}$  is fired if the counter value is within the allowed number of execution cycles interval for  $ts_i$ , the observed conditions of  $ts_{i+1}$  are satisfied and the trigger variable is true. As mentioned before the trigger variable is an extra free observer input that acts as a non-deterministic oracle for triggering transitions. Its purpose is to correctly take into account the existentially quantified *span variables* used in the definition of the LTSL satisfaction relation. Take for instance the specification and the trace, shown in Fig. 4.

```
Test case TC1 :
"trigger example"
// TS1
For [2,?] cycles:
Observe c1;
// TS2
For [2,3] cycles:
Observe c2;
// TS3
For [2,?] cycles:
Observe c3;
```

cycle nbr	0	1	2	3	4	5	6	7	8
$c_1$	1	1	1	1	1	0	0	0	0
$c_2$	0	0	1	1	1	1	0	0	0
$c_3$	0	0	0	0	0	0	1	1	0
eager	TS1		TS2			KO			
trig	0	0	0	0	1	0	1	0	1
nondet	TS1				TS2		TS3		OK

**Fig. 4.** Eager and non-deterministic transitions for LTSL satisfaction

An automaton in which transitions would be fired eagerly as soon as their guard holds would match TS1 over [0,1], TS2 over [2,4] and reject the trace at cycle 5 because  $c_3$  does not hold yet. However, this trace is a model of the specification, because snippet TS1 is satisfied over [0,3], TS2 is satisfied over [3,5] and TS3 is satisfied over [6,7].

By correctly choosing the value of the trigger variable as shown in Fig. 4, we can accept the trace: the transition from  $TS1$  to  $TS2$  can be taken at cycle 2, 3 or 4 since  $c_1$  and  $c_2$  overlap. The only correct choice is cycle 4, so as to be able to match  $TS3$  at cycle 6. The trigger value computation is left in practice to the model-checker used for refuting the observer output. The trigger variable can safely be omitted for transitions linking snippets with mutually exclusive observe conditions, which is often the case in practice.

The body of a state consists of a counter definition and optionally several check label definitions. The counter is initialized at 0, with an increment of 1, and a saturation value is defined by the  $cSat$  function. Each check label is defined as an equality between  $cpt$  and its relative position within  $ts_i$ , thus allowing to locate the place to insert checks in the post-processing phase.

State  $S_n$  encodes the last TS  $ts_n$ . Its error transition is defined as previously, and its accept transition only depends on its own observe conditions since there is no next snippet. Finally, the final state represents the acceptance of all trace snippets in sequence and sets the `Incomplete` flow to false.

### 5 Illustration of the Approach

The LTSL language, translation function, post processing and model-checker harness have been implemented and used on an industrial use case. TPs obtained with the approach proved to be satisfactory with respect to the previous hand-implemented TPs.

We now illustrate the results obtained with this approach when generating a TP from the LTSL specification given in Fig. 3 for the SCADE model depicted in Fig. 2. The SystereL S3 model-checker [2] is used on the instrumented model to refute the observer output, yielding traces for the software main inputs which satisfy the test conditions expressed in the LTSL specification. These traces are then post-processed to be translated into a properly formatted TP extended with check of input values and test outcomes, expressed as a sequential program in the TCL language. The TP is presented graphically in Fig. 5. Please notice that the model-checker generates the shortest possible trace satisfying the test specification, and that even in this case, the generated trace includes some latency. This illustrates the need for “don’t care” bounds in the LTSL language. Figure 7 shows the result of regenerating a TP implementation from an unmodified LTSL specification on a new SCADE model, depicted in Fig. 6, in which evolutions were applied in LLRs other than the one under test. TP maintenance/regeneration of

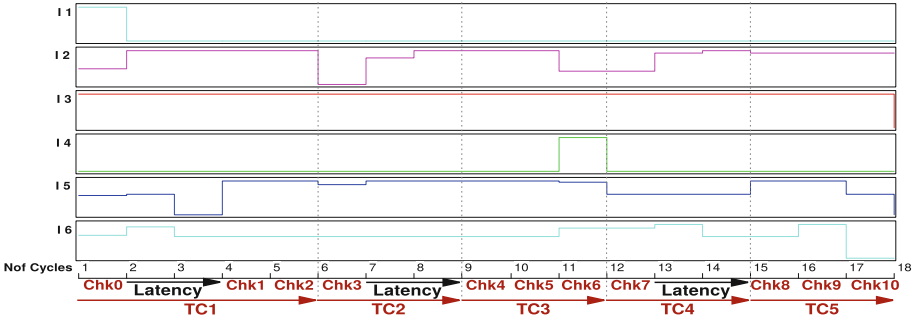


Fig. 5. Time-line of TP for Req1 and SCADE model depicted in Fig. 2.

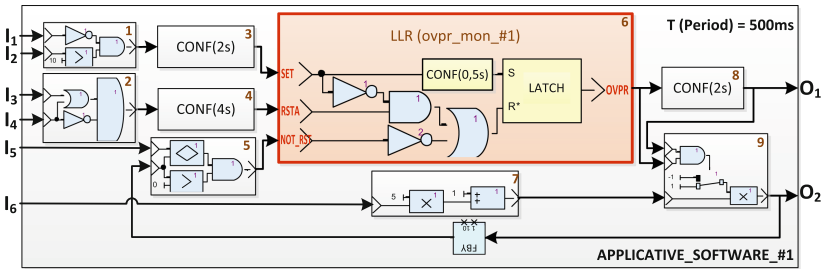


Fig. 6. Evolution of applicative software SCADE model



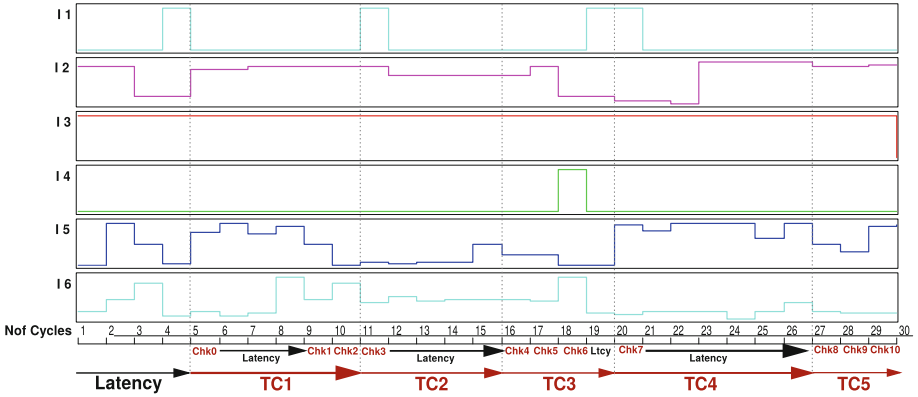


Fig. 7. Time-line of TP corresponding to Req1 and SCADE model depicted in Fig. 6

Tps under model evolution is fully automatic with this method, as long as the LTSL specification remains unmodified.

## 6 Related Work

The Liebherr Test Specification Language semantics and observers generation is inspired from previous works about runtime monitoring of LTL properties [10, 14]. More precisely, we have tried using the *LTL on finite traces* framework [5, 6, 8, 11] to formalize LTSL and generate monitors, by encoding LTSL to LTLf, and then using the LTLf to unbounded LTL embedding proposed in [7], to in turn generate a monitor using the LTL2TGBA tool. However the LTLs trace patterns do not need the full power of the LTLf logic, and monitor generation with LTL2TGBA ran into combinatorial explosion due to having to encode trace snippets as disjunctions of nested pre formulas:  $\circ(c) \vee \circ(\circ(c)) \vee \dots \vee \circ(\dots \circ(c) \dots)$ , which grows quadratically with trace snippet’s duration upper bound. We then took inspiration from recent work on *trace-insensitive LTL monitoring* [9], which proposes an LTL variant extended with counters and deterministic monitor generation. From these earlier works, we distilled the minimal semantics and observer generation approach presented in this paper, to meet our industrial needs. Since these observers are meant for refutation with a model checker, it allowed us to retain some non-determinism in the semantics and observer encoding – through the observer trigger variable, which is beneficial to the size of the resulting observer.

## 7 Conclusion

In this paper, we have presented an approach to assist the generation of TPs. We have defined LTSL as a formal and declarative language to formalize TCs. LTSL

specifications are automatically translated into synchronous SCADE observers, which are used to instrument the model and from which TPs can be generated by leveraging model-checking techniques. This approach was illustrated in this paper by a simplified example. Another example has been given to highlight the robustness of LTSL specification against possible evolutions of the SCADE model under test, as long as the LTSL specification remains unchanged. The pertinence of our work has been demonstrated by using our approach on an actual industrial AMS software. The TPs obtained by this new method are satisfactory, in the sense that they cover the test objectives and implement the oracle checks as required by the test specification.

The perspectives are firstly to prove more formally the adequation between LTSL formal semantics and the generated SCADE observer. Secondly, in order to handle cases in which test cases are not reachable, we need to resort to unbounded model checking to terminate test case generation when objectives are truly unreachable (today only bounded model checking is used and the analysis can diverge in the presence of unreachable objectives). Then, we need to test the approach on more significant numerical use cases, in particular involving floats. For extending the expressive power of LTSL, we would like to allow the positioning checks relative to the end of a trace snippet, even in cases where a “don’t care” duration upper-bound is specified. Finally, the parameterization and the composition of the LTSL TCs to factorize a maximum the LTSL specification. Not addressed in this paper by lack of space is the case of multi-periodic SCADE models scheduled and robustness to model scheduling. We have addressed such models successfully in practice but the details of the multi-periodic LTSL semantics and observer implementation remain to be formalized. Also not addressed in this paper is the question of observability of checked variables. Some of them are only indirectly observable in the program under test, and would require automatic reformulation of the checked conditions.

## References

1. SCADE Suite. <http://www.esterel-technologies.com/products/scade-suite>. Accessed 6 Feb 2018
2. Systereel Smart Solver. <http://www.systereel.fr/innovation/produits/systereel-smart-solver>. Accessed 6 Feb 2018
3. Beneviste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernie, P., de Simone, R.: The synchronous languages 12 years later. *Proc. IEEE* **91**(1), 64–83 (2003)
4. Colaço, J.L., Pagano, B., Pouzet, M.: A conservative extension of synchronous data-flow with state machines (2005)
5. De Giacomo, G., De Masellis, R., Montali, M.: Reasoning on LTL on finite traces: insensitivity to infiniteness (2014)
6. De Giacomo, G., De Masellis, R., Grasso, M., Maggi, F.M., Montali, M.: LTLF and LDLF monitoring: a Technical report (2014)
7. De Giacomo, G., Vardi, M.Y.: Synthesis for LTL and LDL on finite traces (2015)
8. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces (2013)

9. Du, X., Liu, Y., Tiu, A.: Trace-length independent runtime monitoring of quantitative policies in LTL (2015)
10. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification (2013)
11. Fionda, V., Greco, G.: The complexity of LTL on finite traces: hard and easy fragments (2016)
12. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language Lustre. *Proc. IEEE* **79**(9), 1305–1320 (1991)
13. Kurtev, I., Hooman, J., Schuts, M.: Runtime monitoring based on interface specifications. In: Katoen, J.-P., Langerak, R., Rensink, A. (eds.) *ModelEd, TestEd, TrustEd*. LNCS, vol. 10500, pp. 335–356. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-68270-9\\_17](https://doi.org/10.1007/978-3-319-68270-9_17)
14. Leucker, M., Schallhart, C.: A brief account of runtime verification (2009)



# Using Test Ranges to Improve Symbolic Execution

Rui Qiu<sup>1</sup>, Sarfraz Khurshid<sup>1</sup>, Corina S. Păsăreanu<sup>2</sup>,  
Junye Wen<sup>3</sup>, and Guowei Yang<sup>3</sup>(✉)

<sup>1</sup> University of Texas at Austin, Austin, USA  
{ruiqiu,khurshid}@utexas.edu

<sup>2</sup> CMU/NASA Ames, Mountain View, USA  
corina.s.pasareanu@nasa.gov

<sup>3</sup> Texas State University, San Marcos, USA  
{j\_w236,gyang}@txstate.edu

**Abstract.** Symbolic execution is a powerful systematic technique for checking programs, which has received a lot of research attention during the last decade. In practice however, the technique remains hard to scale. This paper introduces *SynergiSE*, a novel approach to improve symbolic execution by tackling a key bottleneck to its wider adoption: costly and incomplete constraint solving. To mitigate the cost, *SynergiSE* introduces a succinct encoding of constraint solving results, thereby enabling symbolic execution to be distributed among different workers while sharing and re-using constraint solving results among them without having to communicate databases of constraint solving results. To mitigate the incompleteness, *SynergiSE* introduces an integration of complementary approaches for testing, e.g., search-based test generation, with symbolic execution, thereby enabling symbolic execution and other techniques to apply in tandem. Experimental results using a suite of Java programs show that *SynergiSE* presents a promising approach for improving symbolic execution.

## 1 Introduction

Symbolic execution is a systematic technique for checking programs, which provides a powerful analysis in principle but remains computationally expensive and hard to scale in practice [11, 17, 21, 23, 25–27, 31, 36, 44, 46]. There are two basic issues with scaling symbolic execution. One, it requires exploring a very large number of program paths. Two, it requires expensive and incomplete solving of constraints, termed path conditions, defined by the program’s operations on its inputs. Researchers developed various approaches to address these issues, and two such approaches form the basis of our work: (1) partitioning of the path exploration problem into several sub-problems in a parallel or distributed setting with minimal communication overhead using several symbolic execution workers [8, 22, 38, 39, 42]; and (2) re-use of constraint solving results within one run of symbolic execution or across different runs in a sequential setting [9, 43, 46, 47].

This paper introduces the *SynergiSE* approach for improving symbolic execution. A key novelty of *SynergiSE* is that it allows obtaining the benefits of the parallel exploration and constraint re-use in tandem without the overhead of communicating databases of constraint solving results or sharing the databases. To our knowledge, none of the existing techniques for parallel symbolic execution [8, 22, 38, 39, 42] share constraint solving results among different workers. While existing techniques for constraint re-use can be applied for the parallel setting in principle, doing so directly requires communicating or sharing constraint databases. Furthermore, *SynergiSE* allows applying complementary test automation techniques, e.g., search-based [15] or feedback-directed random [28] test generation, in tandem with symbolic execution.

At the heart of *SynergiSE* lies a new form of *range* analysis that builds on previous work on ranged symbolic execution [39], which introduced the idea of bounding a run of symbolic execution using a *range* defined by a pair of two *ordered* tests  $(t_1, t_2)$  such that symbolic execution is restricted to program paths that are lexicographically between the path executed by  $t_1$  and the path executed by  $t_2$ . We extend that work by introducing two special kinds of ranges – *feasible ranges* that succinctly encode constraint solving results for lightweight communication among parallel workers, and *unexplored ranges* that succinctly encode unexplored regions of the program’s exploration space.

A *feasible range* is a range such that all paths explored by symbolic execution within that range are known to be feasible, e.g., during an earlier run of symbolic execution during iterative deepening, or during the exploration by another worker in a parallel setting. Thus, a feasible range uses just two inputs to compactly encode constraint satisfaction of all the path conditions for the paths in the range. Given a feasible range, the corresponding constraint satisfaction database can be quickly populated by just building the path conditions for all paths that are within the range – all such paths are feasible by definition – without requiring any additional constraint solving. The populated database can then be efficiently re-used when symbolically executing other, unexplored, parts of the program.

An *unexplored range* is a range that only contains paths that are not explored by existing tests of a program or by a previous partial exploration of the program’s execution paths. Unexplored ranges enable symbolic execution to efficiently reuse existing tests, providing a natural integration between any test generation tool and symbolic execution. Previously generated tests using other tools, such as random testing tools or search-based tools, can be ordered [39] with respect to the specific search order used by the symbolic execution tool employed to define a set of unexplored ranges, which only contain program paths that none of the existing test cover. Efficiently reusing test inputs from another tool for symbolic execution helps speed up analysis, generate more tests that cover yet unexplored program paths, hence increasing code coverage as well as confirming that certain paths not covered previously are actually infeasible. Unexplored ranges enable a new form of integration of symbolic execution with other test generation techniques, which may already have an internal symbolic execution component.

This paper makes the following contributions:

- **Feasible ranges.** We introduce the idea of feasible ranges that compactly encode path condition feasibility for all paths that are explored by symbolic execution with respect to the ranges; consecutively explored in order by symbolic execution.
- **Unexplored ranges.** We introduce the idea of unexplored ranges that compactly encode execution paths that are not covered by existing tests for a program.
- **Framework.** We present *SynergiSE*, a framework for symbolic execution, which enables distributed analysis and constraint reuse in tandem using feasible ranges without communicating or sharing constraint databases, as well as enables symbolic execution to apply in synergy with other test generation techniques using unexplored ranges.
- **Evaluation.** We present an experimental evaluation using standard subjects to show that *SynergiSE* significantly reduces the amount of communication among different symbolic execution workers and enables an effective integration of heuristics-based and systematic approaches for test generation.

## 2 The *SynergiSE* Approach

Figure 1 shows an overview of *SynergiSE*. *SynergiSE* uses two types of ranges to improve symbolic execution. Feasible ranges define only feasible paths while unexplored ranges define only unexplored paths.

To obtain **feasible ranges**, *SynergiSE* takes a program  $P$  and performs a shallow symbolic execution up to a small depth  $d$  ( $d < D$ ), where  $D$  is a user specified depth, and uses infeasible paths to divide symbolic execution tree into a set of ranges which only contain feasible paths of the program  $P$ . *SynergiSE* then assigns each feasible range to a distributed worker who performs a ranged symbolic execution on the paths within the range. The key novelty of feasible ranges is that they enable efficient reuse and sharing of constraint solving results among distributed workers. To obtain **unexplored ranges**, *SynergiSE* uses a complementary test generation tool (e.g., Randoop or EvoSuite) to generate a set of test inputs for the program  $P$ . These tests define a set of unexplored ranges, which only contain paths that are not explored by any existing tests. Similarly, each unexplored range is assigned to a distributed worker to perform ranged symbolic execution. The key novelty of unexplored ranges is that they enable efficient reuse of test cases generated from other complementary test generation tools.

### 2.1 Traditional Ranges

Consider a program  $P$  with  $m$  symbolic inputs  $I = \{I_1, I_2, \dots, I_m\}$ . A bounded symbolic execution over  $P$  explores all the paths up to  $D$ . Our technique assumes that program  $P$  is deterministic and symbolic execution follows a fixed search

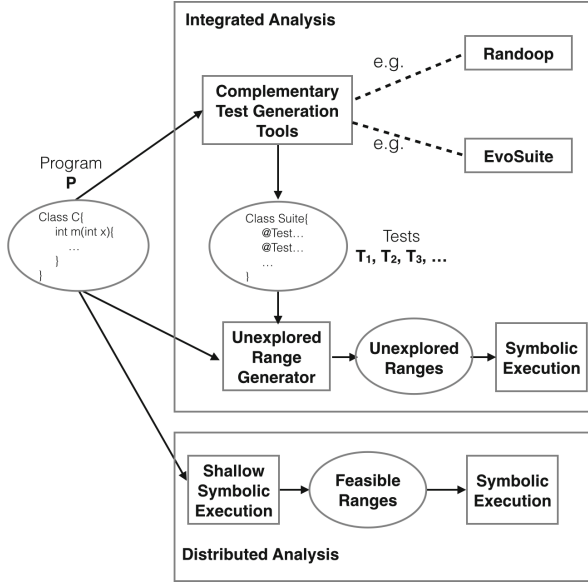


Fig. 1. An overview of *SynergiSE* approach

order, e.g., a standard depth-first search in which the “true” branch of every symbolic conditional instruction is always explored before its “false” branch. Thus, symbolic execution explores an ordered list of paths  $R = [\rho_1, \rho_2, \dots, \rho_n]$  of program  $P$  up to depth  $D$ . For any path  $\rho_i (1 \leq i \leq n)$  in  $R$ , an off-the-shelf solver generates a corresponding *test input*  $T_i$  containing concrete input values for each symbolic input in  $I$ .

Traditional ranges [39] encode subsets of the program state space using test inputs based on the following observations. Any test input succinctly represents an analysis state in symbolic execution. Furthermore, any two test inputs  $T_i$  and  $T_j$  for program  $P$  naturally form an order based on the order of program paths. Specifically, two test inputs  $T_i$  and  $T_j$  drive program  $P$  to follow two paths  $\rho_i$  and  $\rho_j$  respectively. If  $\rho_i$  and  $\rho_j$  are the same path in program  $P$ , we have  $T_i = T_j$ . If  $\rho_i$  is explored before  $\rho_j$  in  $R$ , we have  $T_i < T_j$ . Otherwise, we have  $T_i > T_j$ . In addition, they define a *range*  $r = [\rho_s, \rho_e)$  to be the set of all paths  $\rho$  in  $R$  such that  $\rho_s \leq \rho < \rho_e$ . We refer to this type of range from the previous work as a traditional range.

## 2.2 *SynergiSE* with Feasible Range

Traditional ranges may contain infeasible paths of the program  $P$ . We introduce a new type of range named **Feasible Range**, where each bounded path within the range must be feasible up to a specified search depth. Specifically, a feasible range  $fr = [T_s, T_e, d]$  is a range that contains all paths  $\rho$  of the program  $P$  up to a depth  $d$  that are between  $\rho_s$  and  $\rho_e$  ( $\rho_s \leq \rho \leq \rho_e$ ), where  $\rho_s$  and  $\rho_e$  are the

paths represented by  $T_s$  and  $T_e$  respectively. The paths in a feasible range may be partial (non-terminating) as they are bounded by depth  $d$ . Note that  $T_s$  and  $T_e$  may represent paths other than  $\rho_s$  and  $\rho_e$  when the search depth is larger than  $d$ .

To form feasible ranges, *SynerqiSE* performs a standard bounded symbolic execution up to the search depth  $d$ , generating an ordered list of  $k$  paths  $R = [\rho_1, \rho_2, \rho_3, \rho_u, \rho_4, \dots, \rho_i, \rho_u, \dots, \rho_j, \rho_k]$ , where  $\rho_u$  denotes an unsatisfiable path, and each path in  $R$  is bounded by depth  $d$ . A set of feasible ranges will be  $\{(\rho_1, \rho_3), (\rho_4, \rho_i), \dots, (\rho_j, \rho_k)\}$ . Again, each path  $\rho$  is represented by a corresponding test case. Essentially, unsatisfiable paths serve as dividers for generating feasible ranges. Note that when we increase search depth from  $d$  to  $d+d'$ , each incomplete path in a feasible range will extend to one or more feasible/infeasible paths of depth  $d+d'$ . At this larger depth the original feasible range may no longer contain all feasible paths. Thus each feasible range is associated with a depth  $d$ . Also note that in a feasible range  $fr = [T_s, T_e, d]$ ,  $T_s$  and  $T_e$  can be the same test case when the path represented by them is between two unsatisfiable paths. If there is no infeasible path at depth  $d$ , one can either increase it to larger depths until the program has an infeasible path or any two paths in the program can form a feasible range as long as there is no infeasible path among them up to the depth  $d$ .

A feasible range has an advantage that by definition no unsatisfiable path (up to a depth) can be encountered within the range. This enables several improvements over traditional ranged symbolic execution as described in the following.

1. **Deepening symbolic execution on a feasible range.** When we perform symbolic execution within a feasible range  $[T_s, T_e, d]$  to a larger depth  $D$ , no constraint solving is needed for the depth that is smaller than  $d$  as the paths within depth  $d$  are all feasible thus all corresponding path conditions are bound to be satisfiable. A constraint solver is only needed when we explore the bounded paths with depth that is greater than the original depth  $d$ . Algorithm 1 shows the procedures for performing symbolic execution of the program  $P$  to a larger depth  $D$  (“deepening”) within a feasible range. We begin with the constraint solver “turned off” (Line 5) and whenever the beginning test input  $T_s$  satisfies the path condition and the depth is  $d$ , the execution reaches the range (Lines 10–11). The execution gets out of the range when the path represented by test  $T_e$  is explored (Lines 12–13). Note that  $T_s$  and  $T_e$  may be the same test input. No constraint solver is needed inside the range (Lines 16–17) while we only “turn on” the solver when the exploration reaches larger depths (Lines 14–15). Any paths that are out of the range are ignored (Lines 18–19). Feasible ranges enable reusing constraint solving results for any path conditions that are encountered within the depth  $d$ , reducing the total number of constraint solver calls thus reducing the execution time.
2. **Inferring constraint satisfaction results from feasible ranges.** Feasible ranges also enable inferring all constraint solving results in a program up to a certain search depth. Let us assume that symbolically executing the program



---

**Algorithm 1.** Algorithm for deepening symbolic execution with a feasible range

---

**Input:** Program  $P$ , Feasible range  $fr[T_s, T_e, d]$ , Depth  $D(D > d)$

**Output:** A list of paths  $L_\rho$  for  $P$  bounded by depth  $D$

```

1: List  $L_\rho \leftarrow$  empty list
2:  $inRange \leftarrow false$ 
3:  $depth \leftarrow 0$ 
4:  $i \leftarrow$  first instruction in  $P$  under symbolic execution
5: turn off constraint solver
6: while  $i$  is not the last instruction in  $P$  do
7:   if  $i$  is a conditional instruction then
8:      $depth \leftarrow depth + 1$ 
9:      $pc \leftarrow i$ 's path condition
10:    if  $(\neg inRange) \wedge (depth = d) \wedge (T_s \text{ satisfies } pc)$  then
11:       $inRange \leftarrow true$ 
12:    else if  $inRange \wedge (depth = d) \wedge (T_e \text{ satisfies } pc) \wedge \neg(T_s \text{ satisfies } pc)$  then
13:       $inRange \leftarrow false$ 
14:    else if  $inRange \wedge (depth > d)$  then
15:      turn on constraint solver
16:    else if  $inRange \wedge (depth \leq d)$  then
17:      turn off constraint solver
18:    else if  $\neg inRange \wedge \neg(T_s \text{ satisfies } pc)$  then
19:      backtrack symbolic execution
20:    if  $(depth == D) \vee (i \text{ is a Return instruction})$  then
21:      Test  $T \leftarrow$  solve  $i$ 's path condition
22:      add  $T$  to  $L_\rho$ 
23:     $i \leftarrow$  next instruction
24: return  $L_\rho$ 

```

---

$P$  to a depth  $d$  results in a set of feasible ranges  $FR = \{fr_1, fr_2, \dots, fr_n\}$ . For each feasible range in  $FR$  we can easily infer that all path conditions (up to depth  $d$ ) within it are all satisfiable due to the definition of feasible ranges that no infeasible paths are included. Furthermore, any path constraints that are not in any feasible ranges of  $FR$  are then unsatisfiable. Essentially, one could infer satisfiability results of all path conditions that are within search depth  $d$  from the set of feasible ranges  $FR$ .

The algorithm for inferring constraints' satisfiability in the program  $P$  is similar to Algorithm 1. We perform a guided symbolic execution without using constraint solvers and if a feasible range is entered, all constraints up to depth  $d$  are satisfiable. All constraints encountered after the execution reaches out of a range and before it enters next feasible range are all unsatisfiable. This enables efficient sharing of constraint solving results in a distributed setting.

3. **Sharing constraint solving results among distributed workers.** Constraint satisfiability checking is expensive in symbolic execution hence various forms of results caching are utilized so that satisfiability of constraints encountered in previous analysis can be retrieved without calling a solver, thus reducing the total number of solver calls. For example, Green [43] uses a

Redis in-memory database to store all path constraints of symbolic execution into a key-value store, in which keys are path constraint strings and values are boolean results of satisfiability (either satisfiable or unsatisfiable).

In a typical distributed setting, each computation machine or core or process as a worker performs symbolic execution on its own without any memory sharing with each other. Sharing constraint solving results among them could be challenging. Feasible ranges provide a lightweight approach for sharing constraint satisfiability among distributed workers. Instead of sharing a possibly very large constraint solving results database (e.g., a Green Redis database) among workers, *SynergiSE* allows workers to share feasible ranges and re-generate the constraint satisfiability results. A set of feasible ranges only contain a number of concrete test inputs and they are usually much more compact than path constraint results, stored as strings. From feasible ranges, one could easily infer all path conditions' satisfiability as shown previously.

A direct application of efficiently sharing constraint solving results among workers using feasible ranges is to speed up symbolic execution in a *regression setting*. Assume that we perform symbolic execution to analyze a program every time it is changed, in order to find out if there is any uncaught exceptions or to generate test cases with high coverage. Program changes are often small and incremental. When we symbolically execute the new slightly modified program, many of the constraint solving results from the previous analysis may be reused. In a distribution setting where each worker maintains its own set of stored constraint solving results, a worker that does not have access to other workers' results may miss some number of constraint satisfiability reuse chances. A simple solution is that whenever a worker finishes its assigned part of symbolic execution, it sends its cached constraint solving results to every other worker. However, these shared results could be very large to distribute over network.

A more efficient way is to share the constraint solving results by sharing feasible ranges. Each worker could generate a set of feasible ranges after it finishes symbolic execution on its assigned portion. To further reduce the number of feasible ranges shared among workers, we merge the feasible ranges generated by all workers. The following are the steps to merge and share feasible ranges: (1) each worker finishes its assigned work portion and generates a set of feasible ranges  $FR$ ; (2) each worker sends its feasible ranges  $FR$  to a coordinator which gathers all to form an ordered list of feasible ranges  $[fr_1, fr_2, \dots, fr_k]$  and merges  $fr_i$  and  $fr_{i+1}$  if and only if there is no infeasible paths among the two feasible ranges; (3) the coordinator sends the merged list of feasible ranges to each worker; (4) each worker repopulates all the path condition satisfiability results for the program using the received feasible ranges.

### 2.3 *SynergiSE* with Unexplored Range

Previous work on traditional ranges [39] provides an algorithm for ordering two test inputs  $T_1$  and  $T_2$  based on the order of paths  $R$  that are explored by symbolic execution. The algorithm performs symbolic execution guided by

test inputs  $T_1$  and  $T_2$ , without constraint solver calls. Whenever a path condition explored is satisfied by test  $T_1$  but not by  $T_2$ , one concludes that  $T_1 < T_2$  and vice versa. If both tests drive program  $P$  to the same terminating path with same path condition,  $T_1$  and  $T_2$  follow the same path, i.e.,  $T_1 = T_2$ . We say tests  $T_1$  and  $T_2$  are *distinct* if  $T_1 \neq T_2$ . Starting from a set of test inputs, either from existing test suites or generated by a shallow symbolic execution, one can create a set of distinct tests  $T = \{T_1, T_2, \dots, T_k\}$ , resulting in  $k + 1$  ranges  $[null, T_1), [T_1, T_2), \dots, [T_k, null)$ . Here *null* represents both beginning and ending boundaries of ranges. By distributing each range to a worker that only analyzes the paths within the range, the entire set of paths in program  $P$  is explored.

The limitation of traditional ranges is that they may contain no complete paths between the beginning test  $T_s$  and the ending test  $T_e$ ; this happens when  $T_s$  and  $T_e$  represent consecutive paths  $\rho_s$  and  $\rho_e$  such that  $e = s + 1$  in the list of paths  $R$ . For example, let us assume that symbolically executing the program  $P$  generates an ordered list of paths  $\{\rho_1, \rho_2, \dots, \rho_k\}$  with corresponding tests  $\{T_1, T_2, \dots, T_k\}$ . If an initial set of tests includes  $T_1, T_2$ , and  $T_3$ , four traditional ranges are generated as  $[null, T_1), [T_1, T_2), [T_2, T_3)$ , and  $[T_3, null)$ . Only the last traditional range contains paths of program  $P$  that are not covered by any existing test inputs while the other ranges only contain program paths that are already represented by test inputs. Assuming our goal for symbolic execution is to generate tests that achieve higher path coverage, ranged analysis on first three traditional ranges will not result in any new test cases that could improve coverage. In a parallel symbolic execution setting they would also incur unnecessary cost as more ranges are distributed among workers while there is no extra gain.

To avoid this problem in traditional ranges, we introduce the notion of **unexplored range**,  $ur(T_s, T_e)$ , where  $T_s$  and  $T_e$  encode two non-consecutive paths  $\rho_s$  and  $\rho_e$  ( $e \neq s + 1$ ) of the program  $P$  respectively. For the previous example, tests  $T_1, T_2$ , and  $T_3$  will not form any unexplored ranges as they encode consecutive paths of the program  $P$ . Thus only one unexplored range  $ur(T_3, null)$  is generated.

Simply ordering a set of test inputs does not suffice for generating unexplored ranges. Algorithm 2 shows how to generate a set of unexplored ranges from a set of initial test inputs (denoted as “Unexplored Range Generator” in Fig. 1). We perform a symbolic execution guided by the test inputs, with the constraint solver “turned off”. A symbolic execution tree data structure *seTree* is constructed and the nodes of the tree correspond to a path condition of the program. If any test inputs satisfy a path condition we add that test input to the corresponding node (Lines 8–10). Otherwise if none of the test inputs satisfy a path condition, the state of this path condition is unexplored by any tests thus a mark *UnexploredStatus* is added to the node and we force symbolic execution to backtrack to previous states (Lines 11–13) as we do not explore any state if no test input guides execution there. Essentially, an *UnexploredStatus* stands for either an unsatisfiable path or a path that is not explored by any existing test

---

**Algorithm 2.** Algorithm for generating unexplored ranges from test inputs

---

**Input:** Program  $P$ , A set of test inputs  $T$

**Output:** A set of unexplored range  $UR$

```

1: Set  $UR \leftarrow \emptyset$ 
2: SETree  $seTree \leftarrow$  new symbolic execution tree
3:  $i \leftarrow$  first instruction in  $P$  under symbolic execution
4: while  $i$  is not the last instruction in  $P$  do
5:   if  $i$  is a conditional instruction then
6:      $pc \leftarrow i$ 's path condition
7:     add new node  $N_{pc}$  for  $pc$  to  $seTree$ 
8:     for each Test  $t \in T$  do
9:       if  $t$  satisfies  $pc$  then
10:        attach  $t$  to  $N_{PC}$ 
11:       if  $\neg (\exists t \in T \text{ satisfies } pc)$  then
12:        attach  $UnexploredStatus$  to  $N_{PC}$ 
13:        backtrack symbolic execution
14:    $i \leftarrow$  next instruction
15: List  $L_T \leftarrow$  preorder traversal  $seTree$  to get all leaf nodes
16: for each Element  $e \in L_T$  do
17:   if  $e$  is  $UnexploredStatus$  then
18:      $t_s \leftarrow$  element before  $e$  in  $L_T$ 
19:      $t_e \leftarrow$  element after  $e$  in  $L_T$ 
20:     add  $(t_s, t_e)$  to  $UR$ 
21: return  $UR$ 

```

---

inputs. After guided symbolic execution is over, we traverse  $seTree$  to gather a list of all leaf nodes (Line 15) and form unexplored ranges based on the indices of  $UnexploredStatus$  marks in the list (Lines 16–20). The entire process does not require any call to a constraint solver as test inputs serve as implicit solver. The algorithm is efficient as checking whether a concrete test input satisfies a path condition is much faster than checking the path condition's satisfiability.

Compared to how traditional ranges are formed [39], Algorithm 2 orders test inputs and generates a smaller number of unexplored ranges if there are any consecutive initial test inputs. Furthermore, it enables symbolic execution to efficiently reuse test cases generated by any automated test generation tools, providing a natural integration of symbolic execution and other test generation tools. Specifically, we use a (potentially very large) number of tests generated by a test generation tool, either a random-based test generation tool like Randoop [29] or a search-based software testing tool like EvoSuite [15], to form a set of unexplored ranges, only including paths that are not covered by any existing tests from the tool. Unexplored ranges could help avoiding re-exploring (possibly a large number of) paths that are already covered by tests while traditional ranges re-explore all paths in the program.

### 3 Evaluation

In this section, we present the experiments we have conducted to evaluate the efficiency and effectiveness of *SynergiSE*. In particular, we evaluate:

- *SynergiSE* with feasible ranges compared to traditional ranges (Sect. 3.2);
- *SynergiSE* with unexplored ranges for integrating test generation tools (Sect. 3.3);
- *SynergiSE* with feasible ranges for sharing constraint solving results among workers (Sect. 3.4).

#### 3.1 Implementation and Subjects

We implement *SynergiSE* in the Symbolic PathFinder (SPF) tool [32], using SPF listeners to generate different types of ranges and to steer symbolic execution within a specified range. For distribution, we use a Java implementation [37] of standard Message Passing Interface (MPI) [14] communication system between a coordinator process and a worker process.

We perform the experiments on the Stampede Linux cluster at the Texas Advanced Computing Center (TACC) [1]. TACC provides powerful computation nodes with reliable and fast connectivity. The programs for the coordinator and each worker node are executed on independent processors without memory sharing.

We evaluate our approach on the following Java subjects: *Rational*, *WBS*, *TCAS*, *MerArbiter*, *Dijkstra*, *JDK 1.5 Sorting Algorithms* and *Red-Black Tree Data Structure*, which were used before for evaluating symbolic execution [3, 19, 30, 34, 35, 41, 45, 46]. These subjects contain rich programming constructs like recursion, that are difficult to handle with symbolic execution. All relevant research artifacts, including subjects, test cases generated, etc. are publicly available for download [2].

#### 3.2 *SynergiSE* with Feasible Ranges

We evaluate the performance of *SynergiSE* with feasible ranges compared to traditional ranges. Table 1 shows the results of our experiments. For each subject, we first perform a shallow symbolic execution with an initial depth, and then perform distributed symbolic execution based on the generated ranges to explore all paths of the program. This initial analysis generates a number of test inputs representing (possibly incomplete) paths of the program, which form a set of feasible ranges as described in Sect. 2.2. To compare feasible ranges with traditional ranges, we also create the same number of traditional ranges using the test inputs generated by the initial analysis, such that each of them contains approximately the same number of test inputs. For example, if the initial analysis generates a list of 13 tests  $[T_1, T_2, T_3, \dots, T_{13}]$ , we can create three traditional ranges as  $[null, T_5)$ ,  $[T_5, T_9)$ ,  $[T_9, null]$ , each covering 4 or 5 tests.

**Table 1.** Evaluation of *SynergiSE* with feasible ranges compared to traditional ranges

Subject	Initial Depth	# Ranges	# Workers	Time (s)		# Solver Calls	
				TR	FR	TR	FR
<b>MergeSort</b> SPF Time: 262 # Solver Calls: 10,366	20	6	6	94	74	10,536	10,366
	23	8	6	88	75	10,615	10,366
	26	12	12	51	51	10,822	10,366
	29	16	12	50	51	11,109	10,366
<b>QuickSort</b> SPF Time: 657 # Solver Calls: 25,920	9	6	6	248	228	26,198	25,920
	12	8	6	275	205	26,337	25,920
	13	12	12	147	408	26,574	25,930
	14	16	12	160	262	26,756	25,932
<b>HeapInsert</b> SPF Time: 72 # Solver Calls: 2,590	25	6	6	16	17	2,717	2,590
	28	8	6	21	21	2,729	2,590
	31	12	12	33	12	2,881	2,590
	34	16	12	21	16	3,007	2,590
<b>RedBlack Tree</b> SPF Time: 505 # Solver Calls: 20,462	25	6	6	198	93	20,735	20,462
	28	10	6	195	130	20,933	20,462
	31	12	12	112	91	21,037	20,466
	34	16	12	131	68	21,239	20,470
<b>WBS</b> SPF Time: 443 # Solver Calls: 27,646	10	6	6	96	98	27,827	27,646
	13	8	6	239	90	27,869	27,646
	16	12	12	169	60	27,981	27,646
	19	16	12	202	71	28,098	27,646

For each subject, there are four different experimental settings, where either 6 or 12 workers plus one additional coordinator are used. Table 1 reports the execution time for traditional ranges (“TR”) and feasible ranges (“FR”). It also reports the number of constraint solver calls for each setting. Table 1 shows that the ranged analysis based on both types of ranges speeds up traditional (sequential) symbolic execution (shown as “SPF Time” in the first column), ranging from  $1.6\times$  (FR 12 ranges 12 workers for *QuickSort*) to  $7.4\times$  (FR 16 ranges 12 workers for *RedBlack Tree*) faster, on average about  $4.5\times$  faster. In most cases, feasible ranges outperform traditional ranges, up to  $2.8\times$  faster. This is also reflected by the fewer constraint solver calls for feasible ranges. The few cases where feasible ranges are slower than traditional ranges due to the imbalance in the feasible ranges.

### 3.3 *SynergiSE* for Integrating Test Generation Tools

In *SynergiSE*, unexplored ranges provide an efficient way for symbolic execution to integrate with other complementary test generation tools. To evaluate this,

**Table 2.** Evaluation of *SynergiSE* for Integrating Test Generation Tools

Subject	Tool	Time (s)	# Tests	# TR/# UR	Time (s)			# Solver Calls			# New Tests	Init/New Cov. (%)
					TR	UR	URS	TR	UR	URS		
<b>TCAS</b> SPF Time: 12 # Solver Calls: 678	Randoop	3	1,737	13/2	11	11	10	784	654	654	335	30/93
		6	3,653	13/2	11	11	10	780	654	654	335	30/93
	EvoSuite	3	9	10/8	9	8	10	856	623	623	337	70/93
		6	8	9/9	5	5	10	852	623	623	339	84/93
		9	10	11/11	9	8	9	921	600	600	337	88/93
		30	9	10/10	6	6	10	887	614	614	338	93/93
<b>Rational</b> SPF Time:258 # Solver Calls: 22,294	Randoop	5	4,319	597/597	122	94	197	52,605	17,659	17,706	8,686	89/89
		10	7,719	639/639	109	96	194	54,426	17,406	17,437	8,644	89/89
	EvoSuite	5	7	8/8	172	170	257	22,650	22,170	22,170	9,275	89/89
		10	8	9/9	122	120	256	22,697	22,154	22,154	9,274	89/89
<b>WBS</b> SPF Time:443 # Solver Calls: 27,646	Randoop	5	3,943	1,702/620	211	178	361	31,276	22,599	22,637	12,124	60/66
		10	6,346	2,293/663	224	175	343	30,644	21,533	21,557	11,533	60/66
	EvoSuite	5	9	10/10	137	136	436	28,058	27,518	27,497	13,816	66/66
		10	12	13/13	95	93	435	28,200	27,488	27,485	13,813	66/66
<b>MerArbiter</b> SPF Time: 820 # Solver Calls: 38,944	Randoop	5	2,747	1,238/965	345	267	744	74,952	33,813	33,885	17,046	57/65
		10	5,517	1,970/1,390	447	318	719	95,839	31,306	31,611	16,314	57/65
	EvoSuite	5	4	5/5	420	417	809	39,070	38,902	38,902	18,279	41/65
		10	8	5/5	444	439	811	39,070	38,910	38,910	18,279	41/65
		15	9	9/9	262	262	810	39,223	38,861	38,861	18,275	49/65
		30	18	14/13	192	189	805	39,387	38,819	38,819	18,270	47/65

we choose two popular test generation tools: Randoop [29] and EvoSuite [15]. We conduct experiments on four subjects as listed in Table 2, with several different settings. In each setting, we first run a test generation tool (denoted as “Tool” in Table 2) on a subject for a short period of time (“Run Time (s)”), generating a number of test inputs (“# Tests”). The run time for test generation tools is varied and is increased until the branch coverage no longer increases or up to 30s. These test inputs could form a number of traditional ranges (“# TR”) or unexplored ranges (“# UR”). The numbers of both types of ranges are the same if no consecutive paths exist in the paths represented by the initial test inputs. We use 6 workers plus one coordinator to run distributed symbolic execution with the generated traditional ranges or with unexplored ranges (“TR” and “UR”). To evaluate the effects of distribution, we also use the unexplored ranges in a sequential setting where only one worker works on all ranges sequentially (“URS”). We report the time for the distributed symbolic execution and the number of total constraint solver calls. We also report the number of new tests generated (“# New Tests”) and the comparison of branch coverage achieved by the initial test cases from the complementary test generation tools and the updated coverage after symbolic execution (“Init/New Cov.”). The execution time and number of constraint solver calls of traditional (sequential) symbolic execution using SPF are listed in the first column of Table 2.

From Table 2 we observe that Randoop generated a large number of tests but only a small fraction of them cover unique paths in the program. The number

of unique paths is the number of traditional ranges minus one. As expected, Randoop tends to have lower branch coverage compared to EvoSuite because Randoop is random testing technique while EvoSuite combines search based and dynamic symbolic execution techniques.

For larger subjects *Rational*, *WBS*, and *MerArbiter*, using a large number of test inputs from Randoop speeds up symbolic execution, regardless of whether a distribution setting is used. If we only consider the sequential setting, Randoop results speed up *Rational*, *WBS*, and *MerArbiter* up to  $1.3\times$ . For EvoSuite, the total time including both initial test generation and the time for subsequent symbolic execution is slightly longer than SPF execution time due to the small number of tests generated. In distribution settings, the speedup is significant for both types of ranges, ranging from about  $1.3\times$  (TR for *Rational* with EvoSuite running 3s) to about  $4.7\times$  (UR for *WBS* with EvoSuite running 10s). The average speedup for the three subjects is about  $2.5\times$ .

Furthermore, the number of unexplored ranges is either the same or less than the number of traditional ranges. In all cases, execution time for using unexplored ranges are faster than or the same as using traditional ranges (about  $1.1\times$  faster on average). In cases where the initial set of tests do not achieve the highest branch coverage, symbolic execution generates more tests, which increase the branch coverage; otherwise, symbolic execution verifies that the coverage achieved by the initial test set is the highest possible, i.e., the other branches that are not covered are actually infeasible branches.

### 3.4 *SynergiSE* for Sharing Constraint Solving Results Among Workers

We evaluate sharing constraint solving results among workers in a regression setting. Each subject listed in Table 3 has ten versions: the original program version and its mutants generated by applying first order mutations [20] (e.g., changing a comparison operator or changing a constant) at randomly selected locations in the code. The mutations simulate small program changes that are common in practice. We first perform distributed symbolic execution with feasible ranges for the original program version using 6 workers plus one additional coordinator (“Orig”) and then we perform the same analysis for the mutants with the same 6 workers in four different settings: (1) each worker does not reuse any constraint solving results from the previous analysis on the original program (“No Reuse”); (2) each worker reuses its own stored constraint solving results from the previous analysis on the original program without sharing across workers (“No Share”); (3) each worker sends its own stored constraint solving results, i.e., a Green database, to the coordinator after the first run of symbolic execution on the original program, and the coordinator then merges all databases and sends the merged database back to each worker for reuse (“Share DB”); (4) each worker sends its updated feasible ranges due to ranged symbolic execution, which extend the feasible range initially assigned to the worker, to the coordinator, the coordinator merges all the received feasible ranges to a set feasible ranges for the whole program and sends this set to each worker, and each



**Table 3.** Evaluation of *SynergiSE* for sharing constraint solving results

Subject	Approach	Time (s)			# Reuses/# Solver Calls		# Constraints	Size (KB)	Zipped Size (KB)	Process Time (s)
		Orig	Mutants		Orig	Mutants				
			Only	Total						
<b>WBS</b> Init Depth: 10	No Reuse	98	77	175	0/27,646	0/23,460				
	No Share	40	26	66	27,564/27,646	23,417/23,460				
	Share DB	45	25	70	27,564/27,646	23,417/23,460	10	2	0.5	4
	Share FR	71	26	97	27,564/27,646	23,417/23,460	10	0.31	0.31	29
<b>HeapInsert</b> Init Depth: 25	No Reuse	16	25	41	0/2,590	0/2,706				
	No Share	15	19	34	0/2,590	207/2,706				
	Share DB	19	<b>17</b>	36	0/2,590	<b>1,700/2,706</b>	2,568	484	59	4
	Share FR	20	<b>16</b>	36	0/2,590	<b>1,700/2,706</b>	2,568	0,33	0.32	5
<b>MergeSort</b> Init Depth: 20	No Reuse	74	87	161	0/10,366	0/11,071				
	No Share	63	73	136	166/10,366	506/11,071				
	Share DB	68	<b>59</b>	<b>127</b>	166/10,366	<b>7,498/11,071</b>	9,982	2,079	350	6
	Share FR	85	<b>60</b>	145	166/10,366	<b>7,498/11,071</b>	9,982	27	1.4	20
<b>RedBlack Tree</b> Init Depth: 25	No Reuse	92	91	183	0/20,462	0/20,462				
	No Share	81	78	159	1,228/20,462	1,228/20,462				
	Share DB	86	<b>64</b>	<b>150</b>	1,228/20,462	<b>9,250/20,462</b>	29,174	4,121	737	6
	Share FR	99	<b>63</b>	162	1,228/20,462	<b>9,250/20,462</b>	29,174	93	4.3	18
<b>QuickSort</b> Init Depth: 9	No Reuse	229	220	449	0/25,920	0/27,339				
	No Share	201	197	398	974/25,920	1,064/27,339				
	Share DB	210	<b>107</b>	<b>317</b>	974/25,920	<b>23,037/27,339</b>	24,915	5,781	1,384	6
	Share FR	232	<b>107</b>	<b>339</b>	974/25,920	<b>23,037/27,339</b>	24,915	110	5.5	27
<b>Dijkstra</b> Init Depth: 6	No Reuse	4,387	4,318	8,705	0/5,059	0/5,135				
	No Share	3,907	3,969	7,876	866/5,059	1,135/5,135				
	Share DB	3,990	<b>2,701</b>	<b>6,691</b>	866/5,059	<b>4,829/5,135</b>	4,862	1,256	314	7
	Share FR	4,362	<b>2,573</b>	<b>6,935</b>	866/5,059	<b>4,829/5,135</b>	4,862	26	1.6	29

worker repopulates all constraint solving results for the original program from this set of feasible ranges for reuse (“Share FR”). In the last two settings, each distributed worker has all constraint solving results from symbolic execution of the original program. The difference is that setting (3) shares the whole Green database while setting (4) shares a set of feasible ranges which can be used to repopulate the entire results of the same Green database.

Table 3 reports the symbolic execution time for the original program (“Orig”), for an individual mutant (“Only”), and for both the original program and a mutant (“Total”). For mutants, it shows the average time cost across all the mutants. It also reports the average number of reuses out of the total number of constraint solver calls. In most cases, sharing constraint results among workers yields larger number of reuses of constraint solving results for symbolically executing the mutants (marked in bold), thus saving the total running time. In addition, Table 3 reports the size of the data shared among workers, in form of either whole Green database or the whole set of feasible ranges (“Size (KB)”). We also report the number of constraints in the data set (“# Constraints”). The size of Green database is much larger than the size of feasible ranges, e.g., Green database for *HeapInsert* is about 1464 times larger than its corresponding set of

feasible ranges. We also measure the sizes of compressed Green databases and sizes of compressed feasible ranges (“Zipped Size (KB)”). In all subjects, the compressed Green databases are still significantly larger than the compressed feasible ranges. On the other hand, it takes longer to get feasible ranges repopulated into a Green database than to merge multiple databases into one Green database (as in column “Process Time”).

From Table 3 we observe that sharing feasible ranges often takes longer total time compared to directly sharing Green databases. This is because in our experiment environment (TACC), the overhead difference between sending a set of feasible ranges and sending its corresponding Green database is negligible compared to our symbolic execution cost.

## 4 Related Work

Our approach extends the work on ranged symbolic execution [39] with two new types of ranges. Unexplored ranges enhance the traditional ranges work by efficiently reusing test inputs from other test generation tools. Feasible ranges only contain feasible paths of the program up to a specified search depth and enable efficient sharing constraint solving results among workers. A poster presentation [33] briefly describes the latter. In this paper we give an extensive description and evaluation for both new ranges.

Dini [12] addressed memoization in the context of ranging in the Korat tool [7] for test generation and data structure repair. The work uses invalid ranges which are conceptually a dual to our feasible ranges, but do not encode path conditions since Korat does not build or analyze them.

While our focus in this paper is on approaches based on distributed analysis and constraint re-use, there exist many other approaches for enhancing symbolic execution, including compositional analysis [16, 34, 35], abstraction [6, 40], and the use of program transformations, such as compiler optimizations [10, 13]. We believe our work on feasible ranges and unexplored ranges would also enable these other approaches to apply in tandem.

A survey on various techniques for automated software test case generation can be found in [4]. Our approach enables a natural efficient integration of symbolic execution with other test case generation techniques. We chose two representative tools Randoop [29] and EvoSuite [15] in this domain to evaluate the feasibility of this integration. Randoop is a technique that improves on random test generation by incorporating feedback from executing already created tests. A set of predefined contracts and filters are checked against the execution of any generated test case thus providing guidance over next generation. EvoSuite is a search-based software testing tool that utilizes evolutionary algorithms. It evolves a population of test suites until a solution is found that fulfills the desired coverage or the allocated resources have been exhausted. The evolution of test suites is guided by a fitness function that depicts how much of coverage a test suite has achieved. In practice, our integration approach is not limited to these two test generation tools and can accommodate other test case generation tools as well.

There are many hybrid approaches that combine the strengths of different test generation tools [5, 18, 24]. For instance, Microsoft Pex accepts previously generated test inputs, builds the path constraints along the executions triggered by these inputs and starts further exploration from there. However this may explore many unnecessary paths. In contrast, our approach organizes the test inputs into ranges and explores only two inputs for each range while also being amenable for parallelization.

Several techniques [8, 22, 38, 39, 42] have been proposed for distributing symbolic execution. However none of them address communicating constraints results between distributed workers. This has been identified as one of the main challenges in previous work on Cloud9 [8] which found constraint solving to account for half or more of total symbolic execution time. With our work, we are addressing this challenge by allowing workers to reuse the computation performed by other workers, where the constraint satisfaction results are communicated using a compact representation, which is provided by our feasible ranges.

## 5 Conclusion

We presented *SynergiSE*, a novel synergistic approach for enhancing symbolic execution using test ranges. *SynergiSE* uses feasible ranges to enable distributed analysis with constraint re-use while minimizing the communication of constraint solving results among parallel workers. *SynergiSE* also uses unexplored ranges to enable integration of complementary test generation tools, such as Randoop or EvoSuite, with symbolic execution. This results in higher quality tests and efficient re-use of tests created by other techniques for effective use of symbolic execution to enhance existing testing tools. Experimental results show that *SynergiSE* can significantly reduce the amount of communication among different symbolic execution workers and enables an effective integration of heuristics-based and systematic approaches for test generation. In future work, we plan to evaluate our approach on larger programs as well as to further optimize our algorithms. We also plan to incorporate compositional approaches for better scaling of symbolic execution.

**Acknowledgments.** This work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-1319688, CCF-1319858, CCF-1549161, CCF-1464123, and CNS-1239498).

## References

1. Stampede. <https://www.tacc.utexas.edu/stampede>
2. [http://cs.txstate.edu/~g\\_y10/synergise](http://cs.txstate.edu/~g_y10/synergise)
3. Albert, E., Gómez-Zamalloa, M., Rojas, J.M., Puebla, G.: Compositional CLP-based test data generation for imperative languages. In: Alpuente, M. (ed.) LOPSTR 2010. LNCS, vol. 6564, pp. 99–116. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-20551-4\\_7](https://doi.org/10.1007/978-3-642-20551-4_7)

4. Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., Mcminn, P.: An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.* **86**(8), 1978–2001 (2013)
5. Baars, A.I., Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Tonella, P., Vos, T.E.J.: Symbolic search-based testing. In: ASE 2011, pp. 53–62 (2011)
6. Beyer, D., Lemberger, T.: Symbolic execution with CEGAR. In: Margaria, T., Steffen, B. (eds.) ISO/IEC JTC1/SC22/WG21 N15100. LNCS, vol. 9952, pp. 195–211. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-47166-2\\_14](https://doi.org/10.1007/978-3-319-47166-2_14)
7. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on Java predicates. In: ISSTA 2002, pp. 123–133 (2002)
8. Bucur, S., Ureche, V., Zamfir, C., Candea, G.: Parallel symbolic execution for automated real-world software testing. In: EuroSys 2011, pp. 183–198 (2011)
9. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI 2008, pp. 209–224 (2008)
10. Chen, J., Hu, W., Zhang, L., Hao, D., Khurshid, S., Liu, X., Zhang, L.: Learning to accelerate symbolic execution via code transformation. Under peer-review
11. Clarke, L.A.: A program testing system. In: ACM 1976, pp. 488–491 (1976)
12. Dini, N.: MKorat: a novel approach for memorizing the Korat search and some potential applications. Master’s thesis, University of Texas at Austin (2016)
13. Dong, S., Olivo, O., Zhang, L., Khurshid, S.: Studying the influence of standard compiler optimizations on symbolic execution. In: ISSRE 2015, pp. 205–215 (2015)
14. Message Passing Interface Forum: MPI: a message-passing interface standard. Technical report, Knoxville, TN, USA (1994)
15. Fraser, G., Arcuri, A.: Evolutionary generation of whole test suites. In: QSIC 2011, Los Alamitos, CA, USA, pp. 31–40 (2011)
16. Godefroid, P.: Compositional dynamic test generation. In: POPL 2007, pp. 47–54 (2007)
17. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI 2005, pp. 213–223 (2005)
18. Inkumsah, K., Xie, T.: Evacon: a framework for integrating evolutionary and concolic testing for object-oriented programs. In: ASE 2007, pp. 425–428 (2007)
19. Inkumsah, K., Xie, T.: Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In: ASE 2008, pp. 297–306 (2008)
20. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.* **37**(5), 649–678 (2011)
21. 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](https://doi.org/10.1007/3-540-36577-X_40)
22. Kim, M., Kim, Y., Rothermel, G.: A scalable distributed concolic testing approach: an empirical evaluation. In: ICST 2012, pp. 340–349 (2012)
23. King, J.C.: Symbolic execution and program testing. *Commun. ACM* **19**(7), 385–394 (1976)
24. Lakhotia, K., McMinn, P., Harman, M.: An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *J. Syst. Softw.* **83**(12), 2379–2391 (2010)
25. Li, G., Andreasen, E., Ghosh, I.: SymJS: automatic symbolic testing of Javascript web applications. In: FSE 2014, pp. 449–459 (2014)

26. Liew, D., Cadar, C., Donaldson, A.F.: Symbooglix: a symbolic execution engine for boogie programs. In: ICST 2016, pp. 45–56 (2016)
27. Ma, K.-K., Yit Phang, K., Foster, J.S., Hicks, M.: Directed symbolic execution. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 95–111. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-23702-7\\_11](https://doi.org/10.1007/978-3-642-23702-7_11)
28. Pacheco, C., Ernst, M.D.: Eclat: automatic generation and classification of test inputs. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 504–527. Springer, Heidelberg (2005). [https://doi.org/10.1007/11531142\\_22](https://doi.org/10.1007/11531142_22)
29. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ICSE 2007, Minneapolis, MN, USA, 23–25 May 2007, pp. 75–84 (2007)
30. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. In: PLDI 2011, pp. 504–515 (2011)
31. Păsăreanu, C.S., Mehltz, 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: ISSA 2008, pp. 15–25 (2008)
32. Păsăreanu, C.S., Rungta, N.: Symbolic PathFinder: symbolic execution of Java bytecode. In: ASE 2010, pp. 179–180 (2010)
33. Qiu, R., Khurshid, S., Păsăreanu, C.S., Yang, G.: A synergistic approach for distributed symbolic execution using test ranges. In: ICSE 2017 - Companion, pp. 130–132 (2017)
34. Qiu, R., Yang, G., Păsăreanu, C.S., Khurshid, S.: Compositional symbolic execution with memoized replay. In: ICSE 2015, pp. 632–642 (2015)
35. Rojas, J.M., Păsăreanu, C.S.: Compositional symbolic execution through program specialization. In: BYTECODE 2013 (ETAPS) (2013)
36. 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](https://doi.org/10.1007/11817963_38)
37. Shafi, A., Carpenter, B., Baker, M.: Nested parallelism for multi-core HPC systems using Java. *J. Parallel Distrib. Comput.* **69**(6), 532–545 (2009)
38. Siddiqui, J.H., Khurshid, S.: ParSym: parallel symbolic execution. In: ICSTE 2010, vol. 1, pp. V1-405–V1-409, October 2010
39. Siddiqui, J.H., Khurshid, S.: Scaling symbolic execution using ranged analysis. In: OOPSLA 2012, pp. 523–536 (2012)
40. Slabý, J., Strejček, J., Trtík, M.: Checking properties described by state machines: on synergy of instrumentation, slicing, and symbolic execution. In: Stoelinga, M., Pinger, R. (eds.) FMICS 2012. LNCS, vol. 7437, pp. 207–221. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-32469-7\\_14](https://doi.org/10.1007/978-3-642-32469-7_14)
41. Souza, M., Borges, M., d’Amorim, M., Păsăreanu, C.S.: CORAL: solving complex constraints for symbolic pathfinder. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 359–374. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-20398-5\\_26](https://doi.org/10.1007/978-3-642-20398-5_26)
42. Staats, M., Păsăreanu, C.: Parallel symbolic execution for structural test generation. In: ISSA 2010, pp. 183–194 (2010)
43. Visser, W., Geldenhuys, J., Dwyer, M.B.: Green: reducing, reusing and recycling constraints in program analysis. In: FSE 2012, pp. 58:1–58:11 (2012)
44. Wang, R., Ning, P., Xie, T., Chen, Q.: MetaSymplit: day-one defense against script-based attacks with security-enhanced symbolic analysis. In: USENIX Security 2013, pp. 65–80 (2013)

45. Yang, G., Khurshid, S., Person, S., Rungta, N.: Property differencing for incremental checking. In: ICSE 2014, pp. 1059–1070 (2014)
46. Yang, G., Păsăreanu, C.S., Khurshid, S.: Memoized symbolic execution. In ISSTA 2012, pp. 144–154 (2012)
47. Yang, G., Person, S., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. *ACM Trans. Softw. Eng. Methodol.* **24**(1), 3:1–3:42 (2014)



# Symbolic Execution and Reachability Analysis Using Rewriting Modulo SMT for Spatial Concurrent Constraint Systems with Extrusion

Miguel Romero<sup>(✉)</sup> and Camilo Rocha<sup>(✉)</sup>

Department of Electronics and Computer Science,  
Pontificia Universidad Javeriana, Cali, Colombia  
{miguel.romero, camilo.rocha}@javerianacali.edu.co

**Abstract.** The usual high degree of assurance in safety-critical systems is being challenged by a new incarnation of distributed systems exposed to the presence of hierarchical computation (e.g., virtualization resources such as container and virtual machine technology). This paper addresses the issue of symbolically specifying and verifying properties of distributed hierarchical systems using rewriting modulo SMT, a symbolic approach for rewriting logic that seamlessly combines rewriting modulo theories, SMT solving, and model checking. It presents a rewrite theory  $\mathcal{R}$  implementing a symbolic executable semantics of an algebraic model of spatially constrained concurrent process with extrusion. The underlying constraint system in  $\mathcal{R}$  is materialized with the help of SMT-solving technology, where the constraints are quantifier-free formulas interpreted over the Booleans and integers, and information entailment is queried via semantic inference. Symbolic rewrite steps with  $\rightarrow_{\mathcal{R}}$  capture *all* possible traces from ground instances of the source state to the ground instances of the target state. This approach, as illustrated with some examples in the paper, is well-suited for specifying and proving (or disproving) existential reachability properties of distributed hierarchical systems, such as fault-tolerance, consistency, and privacy.

## 1 Introduction

The widespread availability of virtualization resources such as container and virtual machine technology are marking a new incarnation of distributed systems. This means that the usual high degree of assurance in safety-critical systems is now exposed to the presence of hierarchical computation (and sharing) of information among groups of distributed and concurrent agents. Such a scenario poses new and challenging goals for formal methods that can be met with the help of symbolic analysis techniques and tools, because they can be used to prove correct the functionality offered by the agents for any possible input and communication interleaving. Rewriting modulo SMT [19] is a symbolic specification and verification method for rewriting logic [16] – a general semantic

framework in which many concurrent models of computation can be naturally specified – that seamlessly combines rewriting modulo theories, SMT solving, and model checking. This approach has been previously used in the symbolic specification and verification of safety-critical infinite-state systems that interact with a nondeterministic environment [19], such as NASA’s Plan Execution Interchange Language (PLEXIL) [11] and the CASH scheduling protocol [6].

This paper addresses the issue of symbolically specifying and verifying properties of distributed hierarchical systems using the rewriting modulo SMT approach. It presents a rewrite theory  $\mathcal{R}$  that implements the operational semantics of the model of spatially constrained concurrent process with extrusion (i.e., mobility) in [13, 15]. In that model, information (e.g., knowledge) can be shared in spatially distributed agents that interact with the global system by launching processes (e.g., programs). The scope of an agent is given by a spatial operator indicating where a process resides within the space structure, where it queries and posts information in a local store. The rewriting logic semantics  $\mathcal{R}$  is executable in Maude [7], thus benefiting from formal analysis techniques and tools such as state-space exploration and automata-based LTL model checking. Safety criteria such as consistency (e.g., does a fault propagate to the global system?), fault-tolerance (e.g., when and how does a local store first become inconsistent?), and privacy (e.g., does a store ever gain enough information as to reveal certain information?; do two stores have the same information?) can be automatically queried.

The rewriting logic semantics  $\mathcal{R}$  axiomatizes posting and querying information from/to a local store, parallel composition of processes, recursion, and extrusion. The underlying constraint system is materialized with the help of SMT-solving technology, where the constraints are quantifier-free formulas interpreted over the Booleans and integers, and information entailment is obtained via semantic inference. In particular, system states with  $n$  local stores are represented as symbolic terms of the form  $t(\phi_1, \dots, \phi_n)$ , where  $t$  encodes the hierarchical structure of spaces and each  $\phi_i$  a local store (i.e., a formula under the control of the SMT solver). State transitions are symbolic rewrite steps modulo SMT between state terms. In one rewrite step from  $t(\phi_1, \dots, \phi_n)$  to  $u(\psi_1, \dots, \psi_k)$ , possibly infinitely many instances of  $t$  can be rewritten to instances of  $u$ . In general, a  $n$ -step symbolic rewrite captures *all* possible traces with  $n$  transitions from ground instances of the source state to the ground instances of the target state. This is one of the reasons why rewriting modulo SMT is well-suited for symbolically proving (or disproving) existential reachability properties of distributed hierarchical systems, such as fault-tolerance, consistency, and privacy.

The rewriting logic specification  $\mathcal{R}$  in the syntax of Maude, the formal verification experiments, examples, and proofs can be all found in the technical report [21].

*Outline.* Section 2 presents some preliminaries on rewriting. Section 3 overviews spatial concurrent constraint systems with extrusion. Section 4 introduces the rewriting logic semantics  $\mathcal{R}$  and mechanical proofs about its executability properties. Section 5 explains how existential reachability properties can be automatically proved with  $\mathcal{R}$ . Section 6 concludes the paper.



## 2 Preliminaries

This section briefly explains order-sorted rewriting logic [16], a semantic framework that unifies a wide range of models of concurrency. Maude [7] is a language and tool to support the formal specification and analysis of concurrent systems in rewriting logic.

An *order-sorted signature*  $\Sigma$  is a tuple  $\Sigma = (S, \leq, F)$  with a finite poset of sorts  $(S, \leq)$  and a set of function symbols  $F$  typed with sorts in  $S$ , which can be subsort-overloaded. A *top sort* in  $\Sigma$  is a sort  $s \in S$  such that  $s' \leq s$  for all  $s'$  in the connected component of  $s$ . For  $X = \{X_s\}_{s \in S}$  an  $S$ -indexed family of disjoint variable sets with each  $X_s$  countably infinite, the *set of terms of sort  $s$*  and the *set of ground terms of sort  $s$*  are denoted, respectively, by  $T_\Sigma(X)_s$  and  $T_{\Sigma,s}$ ; similarly,  $T_\Sigma(X)$  and  $T_\Sigma$  denote, respectively, the set of terms and the set of ground terms. A *substitution* is an  $S$ -indexed mapping  $\theta : X \rightarrow T_\Sigma(X)$  that is different from the identity only for a finite subset of  $X$  and such that  $\theta(x) \in T_\Sigma(X)_s$  if  $x \in X_s$ , for any  $x \in X$  and  $s \in S$ . A substitution  $\theta$  is called *ground* iff  $\theta(x) \in T_\Sigma$  or  $\theta(x) = x$  for any  $x \in X$ . The application of a substitution  $\theta$  to a term  $t$  is denoted by  $t\theta$ .

A *rewrite theory* is a tuple  $\mathcal{R} = (\Sigma, E \uplus B, R)$  with: (i)  $(\Sigma, E \uplus B)$  an order-sorted equational theory with signature  $\Sigma$ ,  $E$  a set of equations over  $T_\Sigma$ , and  $B$  a set of structural axioms – disjoint from the set of equations  $E$  – over  $T_\Sigma$  for which there is a finitary matching algorithm (e.g., associativity, commutativity, and identity, or combinations of them); and (ii)  $R$  a finite set of rewrite rules over  $T_\Sigma$ .

Intuitively,  $\mathcal{R}$  specifies a concurrent system whose states are elements of the set  $T_{\Sigma/E \uplus B}$  of  $\Sigma$ -terms modulo  $E \uplus B$  and whose concurrent transitions are axiomatized by the rules  $R$  according to the inference rules of rewriting logic [5]. In particular, for  $t, u \in T_\Sigma$  representing states of the concurrent system described by  $\mathcal{R}$ , a transition from  $t$  to  $u$  is captured by a formula of the form  $t \rightarrow_{\mathcal{R}} u$ ; the symbol  $\rightarrow_{\mathcal{R}}$  denotes the binary rewrite relation induced by  $R$  over  $T_{\Sigma/E \uplus B}$  and  $\mathcal{T}_{\mathcal{R}} = (T_{\Sigma/E \uplus B}, \rightarrow_{\mathcal{R}})$  denotes the *initial reachability model* of  $\mathcal{R}$ . The expressions  $\mathcal{T}_{\Sigma/E \uplus B}$  and  $=_{E \uplus B}$  denote, respectively, the *initial algebra* of  $(\Sigma, E \uplus B)$  and the congruence induced by  $(\Sigma, E \uplus B)$  on  $\Sigma$ -terms.

Appropriate requirements are needed to make an equational theory  $\mathcal{R}$  *admissible*, i.e., *executable* in Maude. It is assumed that the equations  $E$  can be oriented into a set of (possibly conditional) sort-decreasing, operationally terminating, and confluent rewrite rules  $\vec{E}$  modulo  $B$  [7]. For a rewrite theory  $\mathcal{R}$ , the rewrite relation  $\rightarrow_{\mathcal{R}}$  is undecidable in general, even if its underlying equational theory is admissible, unless conditions such as coherence [24] are given (i.e., whenever rewriting with  $\rightarrow_{R/E \cup B}$  can be decomposed into rewriting with  $\rightarrow_{E/B}$  and  $\rightarrow_{R/B}$ ). The admissibility of a rewrite theory  $\mathcal{R}$  ultimately means that its mathematical and execution semantics coincide.

## 3 Spatial Concurrent Constraint Systems with Extrusion

*Concurrent Constraint Programming* (CCP) [22] is a model for concurrency that combines the traditional operational view of process calculi with a *declarative*

view based on logic. Under this paradigm, the concept of *store as valuation* in the von Neumann model is replaced by the notion of *store as constraint* and processes are seen as information transducers. The CCP model is parametric in a *constraint system* (CS) specifying the structure and interdependencies of the partial information that processes can query (*ask*) and post (*tell*) in the *shared store*. Given a signature  $\Sigma$  and a first-order theory  $\Delta$  over  $\Sigma$ , constraints can be thought of as first-order formulae over  $\Sigma$ . The (binary) entailment relation  $\vdash$  over constraints is defined for any pair of constraints  $c$  and  $d$  by  $c \vdash d$  iff the implication  $c \Rightarrow d$  is valid in  $\Delta$ .

**Definition 1 (Constraint Systems).** *A constraint system (CS)  $\mathbf{C}$  is a complete algebraic lattice  $(Con, \sqsubseteq)$ . The elements of  $Con$  are called constraints. The symbols  $\sqcup$ , *true*, and *false* are used to denote the least upper bound (lub) operation, the bottom, and the top element of  $\mathbf{C}$ , respectively.*

The elements of the lattice, the *constraints*, represent (partial) information. A constraint  $c$  can be viewed as an *assertion* (or a *proposition*). The lattice order  $\sqsubseteq$  is meant to capture entailment of information:  $d \sqsubseteq c$ , alternatively written  $c \sqsupseteq d$ , means that the assertion  $d$  represents as much information as  $c$ . Thus,  $d \sqsubseteq c$  may be interpreted as saying that  $c \vdash d$  or that  $d$  can be *derived* from  $c$ . The *least upper bound* (lub) operator  $\sqcup$  represents join of information and thus  $c \sqcup d$  is the least element in the underlying lattice above  $c$  and  $d$ , asserting that both  $c$  and  $d$  hold. The top element represents the lub of all, possibly inconsistent, information, hence it is referred to as *false*. The bottom element *true* represents the empty information.

The authors of [13,15] extend the notion of CS to a distributed and multi-agent scenario where each agent  $i$  has a *space*  $[\cdot]_i$  for storing its local information and for performing computation. Intuitively, the expression  $[c]_i$  asserts that the constraint  $c$  holds within a space attributed to agent  $i$ .

**Definition 2 (Spatial Constraint System with Extrusion [13,15]).** *A  $n$ -agent spatial constraint system with extrusion ( $n$ -SCSE) is a tuple  $(\mathbf{C}, [\cdot]_1, \dots, [\cdot]_n, \uparrow_1, \dots, \uparrow_n)$ , with  $\mathbf{C} = (Con, \sqsubseteq)$  a constraint system,  $[\cdot]_1, \dots, [\cdot]_n$  and  $\uparrow_1, \dots, \uparrow_n$  self-maps over  $Con$ , satisfying for  $1 \leq i \leq n$ :*

- S.1  $[true]_i = true$ ,
- S.2  $[c \sqcup d]_i = [c]_i \sqcup [d]_i$  for each  $c, d \in Con$ , and
- S.3  $\uparrow_i$  is the right inverse of  $[\cdot]_i$ .

Given an  $n$ -SCSE, each  $[\cdot]_i$  is thought as the space (or space function) of the agent  $i$  and each  $\uparrow_i$  is thought as the execution of the given process outside agent's  $i$  space. Property S.1 requires space functions to be strict maps (i.e., bottom preserving) where an empty local space equates to having no knowledge. Property S.2 states that space functions preserve (finite) lubs, and also allows to join and distribute the local information of any agent  $i$ . Property S.3 defines extrusion to be the right inverse of space functions (which is assumed to always exist). If  $n$  is unimportant, then  $n$ -SCSE is simply written as SCSE.

The *spatial concurrent constraint programming with extrusion* (SCCP) calculus is parametric on a SCSE.

**Definition 3 (SCCP Processes [13,15]).** Let  $(C, [\cdot]_1, \dots, [\cdot]_n, \uparrow_1, \dots, \uparrow_n)$  be an  $n$ -SCSE,  $A = \{1, \dots, n\}$  a set of  $n$  agents, and  $V$  an countably infinite set of variables. Consider the following EBNF-like syntax:

$$P ::= \mathbf{0} \mid \mathbf{tell}(c) \mid \mathbf{ask}(c) \rightarrow P \mid P \parallel P \mid [P]_i \mid P \uparrow_i \mid x \mid \mu x.P$$

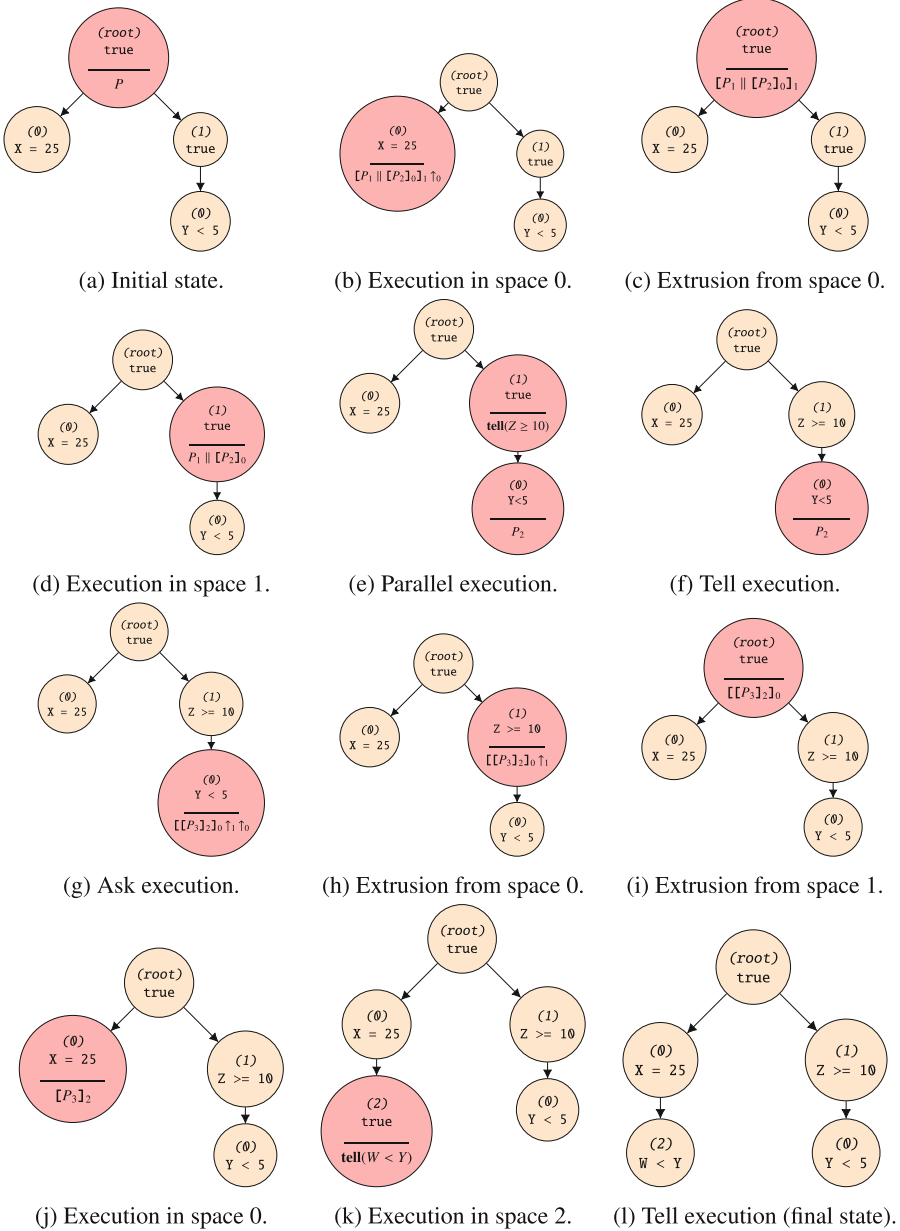
where  $c \in \text{Con}$ ,  $i \in A$ , and  $x \in V$ . An expression  $P$  in the above syntax is a process iff every variable  $x$  in  $P$  occurs in the scope of an expression of the form  $\mu x.P$ . The set of processes of SCCP is denoted by  $\text{Proc}$ .

The SCCP calculus can be thought of as a *shared-spaces* model of computation. Each agent  $i \in A$  has a computational space of the form  $[\cdot]_i$ , possibly containing processes and other agents' spaces. The basic constructs of SCCP are **tell**, **ask**, and parallel composition, and they are defined as in standard CCP [22]. A process **tell**( $c$ ) running in an agent  $i \in A$  adds  $c$  to its local store  $s_i$ , making  $c$  available to other processes in the same space. This addition, represented as  $s_i \sqcup c$ , is performed even if the resulting constraint is inconsistent. The process **ask**( $c$ )  $\rightarrow P$  running in space  $i$  may execute  $P$  if  $c$  is entailed by  $s_i$ , i.e.,  $c \sqsubseteq s_i$ . The process  $P \parallel Q$  specifies the *parallel execution* of processes  $P$  and  $Q$ . A construction of the form  $[P]_i$  denotes a process  $P$  running within the agent  $i$ 's space. Any information that  $P$  produces is available to processes that lie within the same space. The process  $P \uparrow_i$  denotes that process  $P$  runs outside the space of agent  $i$  and the information queried and posted by  $P$  resides in the store of the parent of agent  $i$ . The behavior of a recursive process  $\mu x.P$  is represented by  $P[\mu x.P/x]$ , i.e., every free occurrence of  $x$  in  $P$  is replaced with  $\mu x.P$ .

The operational semantics of SCCP [15] is defined over configurations. A *configuration* is a pair of the form  $\langle P, c \rangle \in \text{Proc} \times \text{Con}$ , where  $P$  is a process and  $c$  is the spatial distribution of information available to it; the set of configurations is denoted by  $\text{Conf}$ . The structural operational semantics of SCCP is captured by the binary transition relation  $\longrightarrow \subseteq \text{Conf} \times \text{Conf}$ , defined by the rules in Fig. 1.

$$\begin{array}{c}
 \frac{}{\langle \mathbf{tell}(c); d \rangle \longrightarrow \langle \mathbf{0}; d \sqcup c \rangle} \text{Tell} \qquad \frac{c \sqsubseteq d}{\langle \mathbf{ask} \ c \ \rightarrow \ P; d \rangle \longrightarrow \langle P; d \rangle} \text{Ask} \\
 \\
 \frac{\langle P; d \rangle \longrightarrow \langle P'; d' \rangle}{\langle P \parallel Q; d \rangle \longrightarrow \langle P' \parallel Q; d' \rangle} \text{Par} \qquad \frac{\langle P; c' \rangle \longrightarrow \langle P'; c' \rangle}{\langle [P]_i; c \rangle \longrightarrow \langle [P']_i; c \sqcup [c']_i \rangle} \text{SP} \\
 \\
 \frac{\langle P[\mu x.P/x]; d \rangle \longrightarrow \gamma}{\langle \mu x.P; d \rangle \longrightarrow \gamma} \text{Rec} \qquad \frac{\langle P; d \rangle \longrightarrow \langle P'; d' \rangle}{\langle [P]_i \uparrow_i; d \rangle \longrightarrow \langle [P']_i \uparrow_i; d' \rangle} \text{Ext}
 \end{array}$$

**Fig. 1.** Structural operational semantics of SCCP.



**Fig. 2.** Execution of process  $P$  and evolution of the SCCP system.

The rules Tell, Ask, Par, and Rec for the basic processes and recursion are the standard ones in CCP. In order to avoid another version of the Par rule for process  $Q$ , parallel composition is assumed to be (associative and) commutative.

The rule **Ext** is a context-dependent definition for extrusion, i.e., it requires a space process (i.e.,  $[\cdot]_j$  with  $j = i$ ) and specifies extrusion in the sense explained before. In the rule **SP**,  $c^i$  represents all the information the agent  $i$  may see or have in  $c$ : namely,  $P$  runs with store  $c^i$ , i.e., with the agent's  $i$  view of  $c$  defined as  $c^i = \bigsqcup \{d \mid [d]_i \sqsubseteq c\}$ . Note that the information  $c'$  added to  $c^i$  by the computation of  $P$  corresponds to the information added by  $[P]_i$  to the space of agent  $i$ . A distinctive property about SCCP is that it allows to even have inconsistent information within spaces; e.g., one agent may have local information  $c$  and the other some local information  $d$  such that  $c \sqcup d = \text{false}$ . This also means that an agent can send inconsistent information to different agents.

As an example of how hierarchical distributed processes evolve with respect to the SCCP's structural operational semantics, consider the sequence of system states depicted in Fig. 2. These states (Figs. 2a–l) correspond to a step-by-step execution from the initial configuration (Fig. 2a), with the process  $P$  defined as follows:

$$P \stackrel{\text{def}}{=} [[P_1 \parallel [P_2]_0]_1 \uparrow_0]_0$$

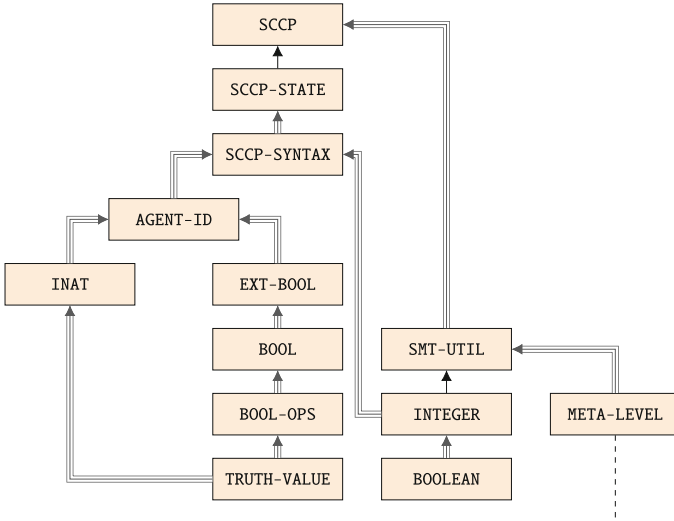
and where:

$$\begin{aligned} P_1 &\stackrel{\text{def}}{=} \mathbf{tell}(Z \geq 10) \\ P_2 &\stackrel{\text{def}}{=} \mathbf{ask}(Y < 20) \rightarrow [[P_3]_2]_0 \uparrow_1 \uparrow_0 \\ P_3 &\stackrel{\text{def}}{=} \mathbf{tell}(W < Y). \end{aligned}$$

## 4 Symbolic Rewriting Logic Semantics

The rewriting logic semantics of a language  $\mathcal{L}$  is a rewrite theory  $\mathcal{R}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}} \uplus B_{\mathcal{L}}, R_{\mathcal{L}})$  where  $\rightarrow_{\mathcal{R}_{\mathcal{L}}}$  provides a step-by-step formal description of  $\mathcal{L}$ 's *observable* run-to-completion mechanisms. The conceptual distinction between equations and rules in  $\mathcal{R}_{\mathcal{L}}$  has important consequences that are captured by rewriting logic's *abstraction dial* [17]. Setting the level of abstraction in which all the interleaving behavior of evaluations in  $\mathcal{L}$  is observable, corresponds to the special case in which the dial is turned down to its minimum position by having  $E_{\mathcal{L}} \uplus B_{\mathcal{L}} = \emptyset$ . The abstraction dial can also be turned up to its maximal position as the special case in which  $R_{\mathcal{L}} = \emptyset$ , thus obtaining an equational semantics of  $\mathcal{L}$  without observable transitions. The rewriting logic semantics  $\mathcal{R}$  sets such an abstraction dial at a position that exactly captures the interleaving behavior of SCCP.

The rewriting logic semantics of SCCP is a rewrite theory  $\mathcal{R} = (\Sigma, E, R)$  with *topsort Sys*. The data types supporting the state structure are defined by the equational theory  $(\Sigma, E)$  and the state transitions are axiomatized by the rewrite rules  $R$ . The constraint system is materialized by an equational theory  $(\Sigma_0, E_0) \subseteq (\Sigma, E)$  of built-ins whose quantifier-free formulas are handled by SMT decision procedures. Figure 3 depicts the module structure of  $\mathcal{R}$ . In this semantics, two modes for importing a module are used, namely, **protecting**



**Fig. 3.** Module structure of the rewriting logic semantics of SCCP.

denoted by a triple arrow  $\Rightarrow$  (meaning no junk and no confusion are allowed on the imported sorts) and **including** denoted by a single arrow  $\rightarrow$  (meaning maybe junk and confusion are allowed on the imported sorts). The reader is referred to [7] for more details about module importation in Maude and their mathematical meaning.

### 4.1 The Constraint System

The materialization of the constraint system in  $\mathcal{R}$  uses SMT solving technology. Given a *many-sorted* (i.e., order-sorted without sort structure) equational theory  $\mathcal{E}_0 = (\Sigma_0, E_0)$  and a set of variables  $X_0 \subseteq X$  over the sorts in  $\Sigma_0$ , the formulas under consideration are in the set  $QF_{\Sigma_0}(X_0)$  of quantifier-free  $\Sigma_0$ -formulas: each formula being a Boolean combination of  $\Sigma_0$ -equation with variables in  $X_0$  (i.e., atoms). The terms in  $T_{\mathcal{E}_0}$  are called *built-ins* and represent the portion of the specification that will be handled by the SMT solver (i.e., semantic data types). Thus, an SMT instance is a formula  $\phi \in QF_{\Sigma_0}(X_0)$  and the initial algebra  $\mathcal{T}_{\mathcal{E}_0^+}$ , where  $\mathcal{E}_0^+$  is a *decidable extension* of  $\mathcal{E}_0$  (typically by adding some inductive consequences and, perhaps, some extra symbols) such that

$$\phi \text{ is satisfiable in } \mathcal{T}_{\mathcal{E}_0^+} \text{ iff } (\exists \sigma : X_0 \longrightarrow T_{\Sigma_0}) \mathcal{T}_{\mathcal{E}_0} \models \phi\sigma.$$

Many decidable theories  $\mathcal{E}_0^+$  of interest are supported by SMT solvers satisfying this requirement (see [19] for details).

The **INTEGER** module implements the equational theory  $\mathcal{E}_0 = (\Sigma_0, E_0)$  of built-ins and the sort **Boolean** defines the data type used to represent the constraints. The topmost concurrent transitions in  $\mathcal{R}$  are then *symbolic* rewrite steps

of state terms with subterms in the set  $T_{\Sigma}(X_0)_{\text{sys}}$  of  $\Sigma$ -terms of sort **Sys** with variables over the built-in sorts in  $\Sigma_0$ .

**Lemma 1.** *The pair  $\mathbf{B} = (QF_{\Sigma_0}(X_0), \models)$  is a constraint system, where  $X_0$  are the variables ranging over the sorts **Boolean** and **Integer**.*

The elements in  $QF_{\Sigma_0}(X_0)$  are equivalence classes of quantifier-free  $\Sigma_0$ -formulas of sort **Boolean** modulo semantic equivalence in  $\mathcal{T}_{\mathcal{E}_0}$  (this technicality guarantees, e.g., the uniqueness of least upper bounds). Therefore, by an abuse of notation, the constraint system  $\mathbf{B}$  has quantifier-free  $\Sigma_0$ -formulas of sort **Boolean** as the constraints and the inverse  $\models$  of the semantic validity relation  $\models$ , w.r.t. the initial model  $\mathcal{T}_{\mathcal{E}_0}$ , as the entailment relation.

In order to use  $\mathbf{B}$  as the underlying constraint system,  $\mathcal{R}$  relies on the current version of Maude that is integrated with the CVC4 [3] and Yices2 [10] SMT solvers. The **SMT-UTIL** module encapsulates this integration, which requires the reflective capabilities of Maude available from the **META-LEVEL** module. The function **entails** implements the semantic validity relation  $\models$  (w.r.t.  $\mathcal{T}_{\mathcal{E}_0}$ ) using the auxiliary functions **check-sat** and **check-unsat** (observe that the sort **Boolean** is different to the usual sort **Bool** for Boolean terms in Maude):

```

op entails      : Boolean Boolean -> Bool.
op check-sat   : Boolean -> Bool.
op check-unsat : Boolean -> Bool.
eq check-sat(B) = metaCheck(['INTEGER], upTerm(B)).
eq check-unsat(B) = not(check-sat(B)).
eq entails(C1, C2) = check-unsat(C1 and not(C2)).

```

The function invocation **check-sat**(**B**) evaluates to true iff **B** is satisfiable; alternatively, it evaluates to false if **B** is unsatisfiable or if the SMT solver times out. The function invocation **check-unsat**(**B**) returns true iff **B** is unsatisfiable. Note that if the constraints **B** are decidable, then **check-unsat** is not only sound but complete. More precisely, if  $\Gamma$  is a finite subset of decidable constraints in  $\mathbf{B}$  and  $\phi$  is also a decidable constraint in  $\mathbf{B}$ , then the following equivalence holds:

$$\mathcal{T}_{\mathcal{E}_0} \models \bigwedge_{\gamma \in \Gamma} \gamma \Rightarrow \phi \quad \text{iff} \quad \text{entails} \left( \bigwedge_{\gamma \in \Gamma} \gamma, \phi \right).$$

## 4.2 System States

The top sort **Sys** is defined in the **SCCP-STATE** module:

```

sort Sys.
op {_} : Cnf -> Sys [ctor].

```

The argument of a state is the configuration of objects representing the setup of the agents and processes in the system. Sort **Cnf** is that of configuration of agents in an object-like notation.

```

sorts Cid Obj Cnf.
subsorts Obj < Cnf.
ops store process : -> Cid.

```

```

op [_,-,-] : Cid Aid Boolean -> Obj [ctor].
op [_,-,-] : Cid Aid SCCPCmd -> Obj [ctor].
op mt : -> Cnf [ctor] .
op __ : Cnf Cnf -> Cnf [ctor assoc comm id: mt].
op {_} : Cnf -> Sys [ctor].

```

The sort **Cnf** represents multisets of terms of sort **Obj**, with set union denoted by juxtaposition. An object is by itself a configuration of objects, namely, the singleton one; constant **mt** denotes the empty configuration and it is the identity of the union operator. There are two types of objects: *process* objects and *store* objects, each represented as a triple  $[-,-,-]$  (as **Obj**). The first two arguments of both a process and a store object are its type (either **process** or **store**, as **Cid**) and its identifier (as **Aid**). The third argument of a process object is the program it is executing (as **SCCPCmd**) and the third argument of a store object is a formula representing the constraint of its corresponding agent (as **Boolean**). The idea is that in any observable state there can be many process objects executing in an agent's space, but there must be exactly one store (i.e., space) object per agent. More precisely, in an observable state, each agent's space is represented by a set of object terms: some encoding the state of execution of all its processes and exactly one object representing its local store.

Process and store objects use a qualified name (sort **Aid**) identifying to which agent's space they belong; this sort is defined in module **AGENT-ID**. Natural numbers (sort **iNat**), in Peano notation and with an equality enrichment [12], are used to specify agents' identifiers. The hierarchical structure of spaces is modeled as a tree-like structure where the root space is identified by the constant **root**. Any other qualified name corresponds to a dot-separated list of agent identifiers, arranged from left to right. That is,  $3 \cdot 1 \cdot \text{root}$  denotes that agent 3 is within the space of agent 1, which in turn is within the top level space of **root**.

```

sorts iNat Aid.
op root : -> Aid.
op _.- : iNat Aid -> Aid.
op 0 : -> iNat [ctor].
op s_ : iNat -> iNat [ctor].
op _- : iNat iNat -> Bool [comm].

```

The processes in SCCP are modeled as *commands* (sort **SCCPCmd**) and are defined in the **SCCP-SYNTAX** module:

```

op 0 : -> SCCPCmd.
op tell_ : Boolean -> SCCPCmd.
op ask_->_ : Boolean SCCPCmd -> SCCPCmd.
op _||_ : SCCPCmd SCCPCmd -> SCCPCmd [assoc comm gather (e E) ].
op <->[_] : iNat SCCPCmd -> SCCPCmd.
op rec(,-) : iNat SCCPCmd -> SCCPCmd.
op xtr(,-) : iNat SCCPCmd -> SCCPCmd.
op v(_): iNat -> SCCPCmd.

```

The argument of a **tell\_** command is a formula (as **Boolean**), namely, the constraint to be added to the corresponding store. The **ask\_->\_** command has a formula (as **Boolean**) and a program (as **SCCPCmd**) as arguments, denoting that if the given formula is entailed by the corresponding store, then the given process is to be executed next. Both arguments of the **\_||\_** command are processes (as **SCCPCmd**). The arguments of the **<->[\_]**, **rec(-, -)**, and **xtr(-, -)** commands are



a natural number (representing the identifier of a child space) and a command to be executed. Note that the syntax of each command is very close to the actual syntax in the SCCP calculus, e.g., constructs of the form  $P \parallel Q$  and  $[P]_i$  in SCCP are represented as SCCPCmd terms of the form  $P \parallel Q$  and  $\langle i \rangle [P]$ , respectively.

*Example 1.* The SCCP space structure in Example 2a can be represented as follows:

```
{ [store, root, true] [store, 0 . root, X:Integer == 25] [store, s 0 . root, true]
  [store, 0 . s 0 . root, Y:Integer < 5] }
```

There are some auxiliary operations defined in the semantics omitted in this paper, e.g., replacing terms in the recursion command; the reader is referred to [21] for these details.

### 4.3 System Transitions

The state transitions in  $\mathcal{R}$  comprise both invisible (given by equations) and observable (given by rules) transitions. There are two types of invisible transitions, namely, one to remove a  $\mathbf{0}$  process from a configuration and other to join the contents of two stores of the same space (i.e., two stores with the same **Aid**). The latter type of transition is especially important because when a new process is spawned in a agent's space, a store with the empty constraint (i.e., **true**) is created for that space. If such a space existed before, then the idea is that the newly created store is subsumed by the existing one (variable **L** is of sort **Aid**, **X** of sort **Cnf**, and **B0**, **B1** of sort **Boolean**):

```
eq { [ process, L, 0 ] X } = { mt X }.
eq { [ store, L, B0 ] [ store, L, B1 ] X } = { [ store, L, B0 and B1 ] X }.
```

The following six rules capture the concurrent observable behavior in the specification (variable **L** is of sort **Aid**, **X** of sort **Cnf**, **B0**, **B1** of sort **Boolean**, **C0**, **C1** of sort **SCCPCmd**, and **N** of sort **iNat**):

```
rl [tell]:
  { [ store, L, B0 ] [process, L, tell B1 ] X }
=> { [ store, L, B0 and B1 ] [ process, L, 0 ] X }.

crl [ask]:
  { [ store, L, B0 ] [ process, L, ask B1 -> C1 ] X }
=> { [ store, L, B0 ] [ process, L, C1 ] X }
if entails(B0, B1).

rl [parallel]:
  { [ process, L, C0 || C1 ] X }
=> { [ process, L, C0 ] [ process, L, C1 ] X }.

rl [space]:
  { [ store, L, B0 ] [ process, L, < N >[ C0 ] ] X }
=> { [ store, L, B0 ] [ process, L, 0 ] [ process, N . L, C0 ] [ store, N . L, true ] X }.

rl [recursion]:
  { [ process, L, rec( N, C0 ) ] X }
=> { [ process, L, replace( C0, N, rec( N, C0 ) ) ] X }.
```

```

r1 [extrusion]:
  { [ process, N . L, xtr( N, C0 ) ] X }
=> { [ process, N . L, 0 ] [ process, L, C0 ] X }.

```

The **[tell]** rule implements the semantics of a process executing a **tell** command by posting the given constraint in the local store and by transforming such a process to the nil process. The **[ask]** rule executes command **C1** when the guard **B1** in **ask B1 -> C1** holds: that is, when **B1** is entailed by the local store **B0**. The **[parallel]** rule implements the semantics for parallel composition of process by spawning the two process in the current space. The **[space]** rule creates a new empty space with agent identifier **N . L** and starts the execution of program **C0** within that space. The **[recursion]** rule defines the semantics of a process executing a **rec** command by using the auxiliary function **replace** (**replace(P, N, C)** substitutes each occurrence of **N** in **P** by **C**). The **[extrusion]** rule executes process **C0** in the parent space and transitions the **xtr** process to be the nil process. Note that a recursion command can lead to non-termination. It is common in SCCP to guard such commands with an ask in order to tame the potential non-termination.

*Example 2.* Consider the following command in the syntax of Example 1:

```

xtr(0, < s 0 >[tell (Z:Integer >= 10) || < 0 >[ask Y:Integer < 20 ->
  xtr(0, xtr(s 0, < s s 0 >[tell (W:Integer < Y:Integer)])])])

```

If this command is executed in the space of agent **0 . root** from the initial state in Example 1 (and depicted in Fig. 2a), it leads to the state

```

{ [store, root, true] [store, 0 . root, X:Integer == 25] [store, s 0 . root, Z:Integer >= 10]
  [store, s s 0 . 0 . root, W:Integer < Y:Integer] [store, 0 . s 0 . root, Y:Integer < 5] }

```

which corresponds to the final state depicted in Fig. 2l.

The soundness and completeness of  $\rightarrow_{\mathcal{R}}$  relative to the SOS of SCCP is stated in Theorem 1 for the fragment of decidable constraints in **B** (e.g., for linear integer arithmetic).

**Theorem 1.** *If  $\langle P, c \rangle$  and  $\langle P', c' \rangle$  are configurations of SCCP with underlying constraint system **B** restricted to decidable formulas, then*

$$\langle P, c \rangle \xrightarrow{*} \langle P', c' \rangle \quad \text{iff} \quad \{ \overline{\langle P, c \rangle} \} \xrightarrow{*}_{\mathcal{R}} \{ \overline{\langle P', c' \rangle} \},$$

where  $\overline{\langle P, c \rangle}$  and  $\overline{\langle P', c' \rangle}$  are an encoding of the corresponding configurations in  $\mathcal{R}$ .

*Proof.* The proof follows by structural induction on the  $\longrightarrow$  and  $\rightarrow_{\mathcal{R}}$  relations.  $\square$

#### 4.4 Admissibility

The admissibility of  $\mathcal{R}$  is obtained with the help of a theory map  $\mathcal{R} \mapsto \mathcal{R}'$  that results, under some assumptions, in a rewrite theory  $\mathcal{R}'$ , equivalent in terms

of admissibility to  $\mathcal{R}$ . The main observation is that dependencies in  $\mathcal{R}$  of non-algebraic data types, such as terms over the built-ins or at the meta-level, are removed while preserving the non-admissibility in of  $\mathcal{R}$  in  $\mathcal{R}'$ . In particular, the map  $\mathcal{R} \mapsto \mathcal{R}'$  consists of the following steps, which make the specification  $\mathcal{R}'$  amenable to mechanical verification in the Maude Formal Environment (MFE) [9]:

- Changing the sort `Bool` in `TRUTH-VALUE` to the sort `iBool` and adjusting the specification to account for this new definition of Boolean values.
- Removing all dependencies in `SMT-UTIL` of the module `META-LEVEL`.
- Introducing a custom if-then-else-fi function symbol in `SCCP-SYNTAX` and adjusting the specification to use this new version instead.

The admissibility proofs are obtained mechanically using the MFE.

**Theorem 2.** *The rewrite theory  $\mathcal{R}'$  is admissible.*

*Proof.* In the MFE, termination of the equational part of  $\mathcal{R}'$  modulo axioms is proved using the Maude Termination Tool, sort-decreasingness and confluence modulo axioms are proved using the Church-Rosser Checker, and coherence is proved using the Maude Coherence Checker:

```
Maude> (ctf SCCP.)
rewrites: 839665 in 187688ms cpu (191102ms real) (4473 rewrites/second)
Success: The functional part of module SCCP is terminating.
...
Maude> (ccr SCCP.)
rewrites: 35929405 in 45540ms cpu (45539ms real) (788963 rewrites/second)
Church-Rosser check for SCCP
  All critical pairs have been joined.
  The specification is locally-confluent.
  The module is sort-decreasing.
...
Maude> (cch SCCP.)
rewrites: 8004447 in 7844ms cpu (7843ms real) (1020454 rewrites/second)
Coherence checking of SCCP
  All critical pairs have been rewritten and no rewrite with rules can
  happen at non-overlapping positions of equations left-hand sides.
```

Finally, the admissibility of  $\mathcal{R}$  can be obtained from the admissibility of  $\mathcal{R}'$  under the assumption that Maude's `META-LEVEL` module is admissible. In particular, it is required that the meta-level functionality used for querying the SMT solver is correct.

**Corollary 1.** *If Maude's `META-LEVEL` is admissible, then  $\mathcal{R}$  is admissible.*

## 5 Symbolic Reachability Analysis

Given the state terms  $t(\phi_1, \dots, \phi_n)$  with  $n$  stores and  $u(\psi_1, \dots, \psi_k)$  with  $k$  stores, the existential reachability question of whether there is a ground substitution  $\theta$  and concrete states  $t' \in t(\phi_1, \dots, \phi_n)\theta$  and  $u' \in u(\psi_1, \dots, \psi_k)\theta$  such that  $t' \xrightarrow{*}_{\mathcal{R}} u'$  is of special interest for many safety properties. For example,  $u'$  can represent a 'bad state' and the goal is to know if reaching such a state is possible. It is important to mention that the approach presented in this section for reachability analysis mainly relies on Maude's `search` command, but it can be easily extended to be used with Maude's LTL Model Checker.

## 5.1 Fault-Tolerance and Consistency

Fault tolerance is the property that ensures a system to continue operating properly in the event of the failure; consistency means that a local failure does not propagate to the entire system. In  $\mathcal{R}$ , this means that if a store becomes inconsistent, it is not the case that such an inconsistency spreads to the entire system. Of course, inconsistencies can appear in other stores due to some unrelated reasons. Finding an inconsistent store can be logically formulated by the following model-theoretic satisfaction instance:

$$\mathcal{T}_{\mathcal{R}} \models (\exists \vec{x}, i \in [1..k]) t(\phi_1, \dots, \phi_n) \xrightarrow{*}_{\mathcal{R}} u(\psi_1, \dots, \psi_k) \wedge \text{unsat}(\psi_i).$$

Answering this query in the positive means that from some initial state satisfying the pattern  $t(\phi_1, \dots, \phi_n)$ , a state can be reached in which a store becomes inconsistent.

Such queries can be easily implemented with the help of  $\mathcal{R}$  and the rewriting modulo SMT approach by using Maude's `search` command. As an example, consider the following `search` command:

```
search in SCCP :
  { [store, root, true] [store, 0 . root, X:Integer == 25] [store, s 0 . root, true]
    [store, 0 . s 0 . root, Y:Integer < 5]
    [process, 0 . root, xtr(0,< s 0 >[tell (Z:Integer >= 10) || < 0 >[ask Y:Integer < 20 ->
      xtr(0,xtr(s 0,< 0 >[< s s 0 >[tell (W:Integer < Y:Integer)]])])]) ] }
=>* { [store, A:Aid, B:Boolean, B0:Boolean] C:Cnf } such that check-unsat(B0:Boolean) .
```

Note that a store is inconsistent if it is unsatisfiable, thereby checking whether a store is inconsistent can be accomplished with the function `check-unsat`. This command does not find an inconsistent store in any of the reachable states. However, it is possible to make a store inconsistent, e.g., by substituting `tell(Z >= 10)` with `tell(Z >= 10) || tell(Z == 9)`:

```
Solution 1 (state 14)
states: 15 rewrites: 678 in 76ms cpu (76ms real) (8921 rewrites/second)
C:Cnf --> [process,s 0 . root,< 0 >[ask Y:Integer < 20 ->
  xtr(0,xtr(s 0,< 0 >[< s s 0 >[tell (W:Integer < Y:Integer)]])]) ]
  [store,root,true] [store,0 . root,X:Integer == (25).Integer]
  [store,0 . s 0 . root,Y:Integer < 5]
A:Aid --> s 0 . root
B0 --> Z:Integer >= (10).Integer and Z:Integer == (9).Integer
...
```

There are 55 reachable states (from the initial state) and 16 of them have an inconsistent store. Note that, even though the inconsistency appears for the first time in state 14, the system evolves until no more processes can be performed. It is possible to verify that there are states with consistent and inconsistent stores at the same time by slightly modifying the search command.

## 5.2 Knowledge Inference

Knowledge inference refers to acquiring new knowledge from existing facts. In the setting of  $\mathcal{R}$ , this means that starting from an initial state, at some point,

an agent has gained enough information to infer – from the rules of first-order logic – new facts:

$$\mathcal{T}_{\mathcal{R}} \models (\exists \vec{x}, i \in [1..k]) t(\phi_1, \dots, \phi_n) \xrightarrow{*}_{\mathcal{R}} u(\psi_1, \dots, \psi_k) \wedge \psi_i \Rightarrow \tau,$$

where  $\tau$  is the formula representing the new fact. Answering the above query in the positive means that from some initial state satisfying the pattern  $t(\phi_1, \dots, \phi_n)$  there is at least an agent as part of a configuration satisfying the pattern  $u(\psi_1, \dots, \psi_k)$  that has enough information to infer  $\tau$ . As an example, consider the following **search** command:

```
search in SCCP :
  { [store, root, true] [store, 0 . root, X:Integer === 25] [store, s 0 . root, true]
    [store, 0 . s 0 . root, Y:Integer < 5]
    [process, 0 . root, xtr(0,< s 0 >[tell (Z:Integer >= 10) || < 0 >[ask Y:Integer < 20 ->
      xtr(0,xtr(s 0,< 0 >[< s s 0 >[tell (W:Integer < Y:Integer)]))]]])] }
=>* { [store, A:Aid, B0:Boolean] C:Cnf } such that entails(B0:Boolean, Y:Integer > 9).
```

It checks if there is a state, reachable from the given initial state, in which some store logically implies  $Y > 9$ . This query does not find a solution.

### 5.3 Same Knowledge

Formally, this type of reachability query can be specified for  $\mathcal{R}$  as follows:

$$\mathcal{T}_{\mathcal{R}} \models (\exists \vec{x}, i, j \in [1..k]) t(\phi_1, \dots, \phi_n) \xrightarrow{*}_{\mathcal{R}} u(\psi_1, \dots, \psi_k) \wedge \psi_i \Leftrightarrow \psi_j \wedge i \neq j,$$

where  $\Leftrightarrow$  denotes logical equivalence. As an example, consider the following Maude **search** command, querying for two stores having the same information when they are non-empty:

```
search in SCCP :
  { [store, root, true] [store, 0 . root, X:Integer === 25] [store, s 0 . root, true]
    [store, 0 . s 0 . root, Y:Integer < 5]
    [process, 0 . root, xtr(0,< s 0 >[tell (Z:Integer >= 10) || < 0 >[ask Y:Integer < 20 ->
      xtr(0,xtr(s 0,< 0 >[< s s 0 >[tell (W:Integer < Y:Integer)]))]]])] }
=>* { [store, A0:Aid, C0:Boolean] [store, A1:Aid, C1:Boolean] C:Cnf }
  such that entails(C0:Boolean, C1:Boolean) and entails(C1:Boolean, C0:Boolean) and
  C1:Boolean /= true.
```

Using Fig. 21 is easy to check that it is never the case that there are two stores with the same information, which agrees with the (omitted) output of Maude.

## 6 Related Work and Concluding Remarks

Rewrite-based executable semantics of process-based formalisms have been proposed before in the realm of rewriting logic and Maude (see, e.g., [4, 8, 23]). They are part of a larger set of formal interpreters developed over the years that have helped in exploring the features of rewriting logic as a semantic framework. The work presented here is a significant extension of the preliminary work initiated in [20]. In particular, the present work adds support for the recursion and extrusion primitives present in SCCP. Other related work in [2] presents  $K$ -stores, a

system for SCCP implemented in Prolog as a constraint interpreter. Two important differences with their work are: (i) extrusion is supported by  $\mathcal{R}$  and (ii) the general approach for symbolic specification and reachability analysis supported by  $\mathcal{R}$  thanks to Maude and its related formal verification tools.

This paper presented a symbolic rewriting logic semantics – based on the rewriting modulo SMT approach – of SCCP [14, 15], an extension of the CCP model [22] with spaces and extrusion. The executable rewriting logic semantics implements the structural operational semantics of SCCP by materializing the underlying constraint system with SMT-based technology. As such, it offers a complete and sound decision procedure for symbolic reachability analysis of existential formulas in SCCP, that can be automatically mechanized using Maude. Examples have been used to illustrate the main concepts and the rewriting logic semantics  $\mathcal{R}$ . The novel idea of combining term rewriting and constrained data structures, as it is the case in  $\mathcal{R}$ , is an active area of research using the rewriting modulo SMT approach [19]. Ultimately, this approach strengthens with symbolic support the wealth of techniques and tools that can be used to symbolically specify and analyze safety-critical systems in Maude.

As future work, extensions of SCCP with real-time and probabilities are a promising line of research. Moreover, providing the rewriting logic semantics of such extensions could lead to interesting case studies for Real-Time Maude [18] and PMaude [1]. Finally, new case studies with applications to emergent systems such as cloud computing and social networks should be pursued with the help of the rewriting logic semantics presented in this work.

**Acknowledgments.** The authors would like to thank C. Rueda and F. Valencia for the fruitful discussions on these ideas, and the anonymous referees for their comments that helped in improving the paper. The first author was partially supported by Colciencias' Convocatoria 761 Jóvenes Investigadores e Innovadores 2016.

## References

1. Agha, G., Meseguer, J., Sen, K.: PMaude: rewrite-based specification language for probabilistic object systems. *Electron. Notes Theor. Comput. Sci.* **153**(2), 213–239 (2006)
2. Barco, A., Knight, S., Valencia, F.D.: K-Stores: a spatial and epistemic concurrent constraint interpreter. In: 21st Workshop on Functional and Constraint Logic Programming (WFLP2012), Nagoya, Japan. *Informal Proceedings* (2012)
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: 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](https://doi.org/10.1007/978-3-642-22110-1_14)
4. Braga, C., Meseguer, J.: Modular rewriting semantics in practice. *Electron. Notes Theor. Comput. Sci.* **117**, 393–416 (2005)
5. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. *Theor. Comput. Sci.* **360**(1–3), 386–414 (2006)
6. Caccamo, M., Buttazzo, G., Sha, L.: Capacity sharing for overrun control. In: 21st IEEE Real-Time Systems Symposium, pp. 295–304. IEEE (2000)

7. Clavel, M. (ed.): All about Maude - a High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic. Lecture Notes in Computer Science, vol. 4350. Springer, Berlin (2007). <https://doi.org/10.1007/978-3-540-71999-1>
8. Degano, P., Gadducci, F., Priami, C.: A causal semantics for CCS via rewriting logic. *Theor. Comput. Sci.* **275**(1–2), 259–282 (2002)
9. Durán, F., Rocha, C., Álvarez, J.M.: Towards a Maude formal environment. In: Agha, G., Danvy, O., Meseguer, J. (eds.) *Formal Modeling: Actors, Open Systems, Biological Systems*. LNCS, vol. 7000, pp. 329–351. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-24933-4\\_17](https://doi.org/10.1007/978-3-642-24933-4_17)
10. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) *CAV 2014*. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-08867-9\\_49](https://doi.org/10.1007/978-3-319-08867-9_49)
11. Estlin, T., Jonsson, A., Pasareanu, C., Simmons, R., Tso, K., Verma, V.: Plan execution interchange language (PLEXIL). Technical report TM-2006-213483, NASA, April 2006
12. Gutiérrez, R., Meseguer, J., Rocha, C.: Order-sorted equality enrichments modulo axioms. *Sci. Comput. Program.* **99**, 235–261 (2015)
13. Guzmán, M., Haar, S., Perchy, S., Rueda, C., Valencia, F.D.: Belief, knowledge, lies and other utterances in an algebra for space and extrusion. *J. Log. Algebr. Methods Program.* **86**(1), 107–133 (2017)
14. Guzmán, M., Valencia, F.D., Herbstritt, M.: On the expressiveness of spatial constraint systems. Technical report, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany (2016)
15. Knight, S., Palamidessi, C., Panangaden, P., Valencia, F.D.: Spatial and epistemic modalities in constraint-based process calculi. In: Koutny, M., Ulidowski, I. (eds.) *CONCUR 2012*. LNCS, vol. 7454, pp. 317–332. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-32940-1\\_23](https://doi.org/10.1007/978-3-642-32940-1_23)
16. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.* **96**(1), 73–155 (1992)
17. Meseguer, J., Roşu, G.: The rewriting logic semantics project: a progress report. *Inf. Comput.* **231**, 38–69 (2013)
18. Ölveczky, P., Meseguer, J.: Real-time maude: a tool for simulating and analyzing real-time and hybrid systems. *Electron. Notes Theor. Comput. Sci.* **36**, 361–382 (2000)
19. Rocha, C., Meseguer, J., Muñoz, C.: Rewriting modulo SMT and open system analysis. *J. Log. Algebr. Methods Program.* **86**(1), 269–297 (2017)
20. Romero, M.: Una semántica ejecutable en lógica de reescritura para programación espacial concurrente por restricciones (SCCP). Trabajo de grado 001/522, Escuela Colombiana de Ingeniería Julio Garavito, Bogotá, Colombia, January 2017
21. Romero, M., Rocha, C.: Reachability analysis for spatial concurrent constraint systems with extrusion (2017). <http://camilorocha.info/publications>
22. Saraswat, V.: *Concurrent Constraint Programming*. Logic programming. MIT Press, Cambridge (1993)
23. Verdejo, A., Martí-Oliet, N.: Two case studies of semantics execution in Maude: CCS and LOTOS. *Formal Methods Syst. Des.* **27**(1–2), 113–172 (2005)
24. Viry, P.: Equational rules for rewriting logic. *Theor. Comput. Sci.* **285**(2), 487–517 (2002)



# Experience Report: Application of Falsification Methods on the UxAS System

Cumhur Erkan Tuncali<sup>1</sup>✉ , Bardh Hoxha<sup>2</sup> , Guohui Ding<sup>3</sup>,  
Georgios Fainekos<sup>1</sup>, and Sriram Sankaranarayanan<sup>3</sup> 

<sup>1</sup> Arizona State University, Tempe, USA  
{[etuncali](mailto:etuncali@asu.edu), [fainekos](mailto:fainekos@asu.edu)}@asu.edu

<sup>2</sup> Southern Illinois University, Carbondale, USA  
[bhoxha@siu.edu](mailto:bhoxha@siu.edu)

<sup>3</sup> University of Colorado, Boulder, USA  
{[guohui.ding](mailto:guohui.ding@colorado.edu), [sriram.sankaranarayanan](mailto:sriram.sankaranarayanan@colorado.edu)}@colorado.edu

**Abstract.** In this report, we present our experiences in applying falsification methods over the Unmanned Systems Autonomy Services (UxAS) system. UxAS is a collection of software modules that enables complex mission planning for multiple vehicles. To test the system, we utilized the tool S-TaLiRo to generate mission scenarios for both UxAS and the underlying vehicle simulators, with the goal of finding behaviors which do not meet system specifications.

## 1 Introduction

Testing and verification of Cyber-Physical Systems (CPS) with respect to their functional or safety requirements is a critical and difficult problem. The difficulty for testing mainly arises from the fact that the generally large input and state spaces of most complex systems make it challenging to identify the values of the inputs and the initial system states which will lead to unexpected behaviors.

Among different testing methodologies, requirements-based boundary-value testing is an approach where the system is tested for the boundary values extracted from the requirements. Although it is a very widely used approach in the industry for testing safety-critical systems, it does not cover the input space of the CPS well. Hence, it may fail to find the failure cases which are not around the boundaries of the requirements. On the other hand, fuzzing [10], where the tests are randomly sampled from the input space of the system, provides a better coverage of the input space. However, the input space is generally infinitely large, especially when it is on real-valued inputs. If an unexpected behavior for the system is caused by a small region in the input space, then, in general, there is a very small probability to hit that small region with randomly

---

This research was supported by the Summer of Innovation 2017 program organized by AFRL and Wright Brothers Institute in Dayton, OH.



generated test cases. Optimization-based test generation/falsification approaches utilize global optimization methods to guide the tests towards a possibly small region in the input space that lead to an incorrect system behavior. Falsification can be defined as the task of discovering counterexamples, i.e., the system behaviors that do not satisfy the given safety or functional requirements.

In this work, we use optimization-based falsification for identifying the conditions that cause an unexpected system behavior. In particular, we used S-TALIRO which is a robustness-guided automatic test case generation tool [4, 9]. S-TALIRO simulates the system with generated input signals and computes a robustness value for the simulated system trajectory. The robustness value is basically a measure of how close is a system trajectory to a set of unsafe behaviors [5]. While a positive robustness value indicates that the trajectory satisfies the system requirements, a negative robustness value means that the trajectory does not satisfy (falsify) at least one system requirement. After computing the robustness value, S-TALIRO utilizes stochastic optimization techniques [1, 8] to update test cases in order to minimize the robustness value. The search continues until it finds a negative robustness value, i.e., a system trajectory falsifying (failing to satisfy) the requirements, or until it exceeds the maximum number of simulations. The stochastic nature of S-TALIRO helps it to obtain a better coverage of the input space of the system compared to the boundary-value testing, while the robustness-guided search approach helps it to smartly guide the tests towards risky areas. This allows exploring the failure cases with a smaller number of simulations compared to random testing, or given a finite time, finding more failure cases than the random testing.

## 2 Problem Statement

UxAS is a publicly available task automation software for Unmanned Aerial Vehicles (UAVs), designed as a set of modular services by the US Air Force Research Laboratory (AFRL) [3]. UxAS computes optimal or close-to-optimal execution plans for a given set of tasks for multiple UAVs and allows cooperative decision making between the UAVs. The search and surveillance tasks UxAS can handle include *point inspection*, *line (path) search*, *area search*, *spiral search* and *sector search* (Table 1) as described by Kingston et al. [6]. The UxAS distribution contains some example scenarios that can be used as a base for these tasks. However, scenarios involve numerous parameters that are specific to an individual mission. Additionally, the missions are carried out under *operating region* constraints that describe *Keep In* regions that a particular aircraft must always remain inside and *Keep Out* regions that an aircraft must keep out of. Finally, the dynamics of the flight are subjected to wind and GPS disturbances.

*Formal System Requirements:* In order to mathematically evaluate whether a trajectory of the system satisfies its requirements, we need formal, mathematical representations of the system requirements. We utilize Metric Temporal Logic (MTL) specifications to formally express the system requirements [7]. MTL

**Table 1.** Examples of parameters describing various mission types in UxAS.

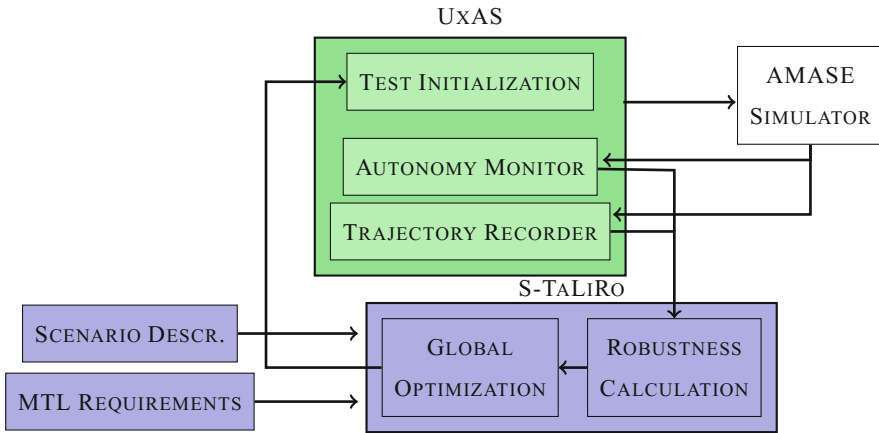
Mission type	Parameters
Point search	GPS coordinate, view angles distance bounds
Line search	Line coordinates, view angles max distance
Area search	Region, desired resolution, angles

extends common temporal operators such as *Finally*, *Globally* and *Until* with time intervals that restrict how these operators are applied to a timed trace. We refer the reader to the original paper by Koymans [7] or our earlier work [1] for a detailed exposition of MTL. Using MTL, we can eliminate ambiguity in the requirements, and we can mathematically reason about the system behavior with respect to its requirements.

Thus, the overall problem is as follows:

INPUTS: Mission parameter ranges, operating region parameter ranges, wind and GPS disturbance parameters, additional MTL requirements.

OUTPUT: Concrete parameter values and operating regions, wind and GPS disturbance patterns so that the resulting plan executions violate mission requirements, operating region requirements or additional MTL constraints.



**Fig. 1.** An overview of the various components involved in the test generation setup for UxAS

### 3 Test Generation for UxAS

A basic overview of our test generation environment architecture is illustrated in Fig. 1. In this section, we will briefly explain the main blocks of this environment.

*AMASE Simulator:* OpenAMASE is an openly available aircraft dynamics simulation toolset developed by the US Air Force Research Laboratory [2]. It communicates with UxAS through a middleware interface and implements simulators for UAV platforms and disturbances such as wind. We have additionally incorporated GPS disturbances into this framework. OpenAMASE is initialized by providing descriptions of the various air vehicles and their initial positions. Next, the UxAS system provides a series of waypoints to the aircrafts to follow. AMASE roughly simulates the behavior of the Piccolo autopilot over these waypoints and periodically publishes aircraft positions and headings back to UxAS as the simulation progresses.

*Scenario Description:* Scenarios are described externally through an XML file that is read and transmitted to the UxAS system through a middleware layer. The message to the UxAS system contains mission information that includes aircraft descriptions and configurations, initial aircraft states, target tasks, Keep In/Out zone parameters and weather conditions which can change during the simulation. In our approach, we mainly generate new test cases by modifying the parameters defined in these messages. We add additional fields to the XML structure to define the ranges of the parameters.

*S-TALiRO/UxAS Interface:* The testing process is started by a Matlab script. This startup script reads the scenario description XML files and extracts the ranges for the variable parameters. It further extracts the information on the path search tasks and the operating zones. If there is a path search task to be randomized, a random path for the task is generated. Similarly, for any keep out zone, it randomly places the Keep Out zone around the path such that the keep out zone does not intersect with the path. The MTL requirements for the system are also specified in the startup script.

After basic configuration is read, S-TALiRO is called with the parameter ranges, the ranges for the coordinates of the Keep Out Zones and the system requirements. Figure 1 gives an overview of our test generation approach. After it starts, S-TALiRO randomly samples from the parameter ranges, and communicates with the test services located in the UxAS over TCP/IP sockets to send the sampled parameter values and to receive the system trajectory at the end of the simulation. The received trajectories are used for computing the robustness value for the current execution. The sampled values are updated, and the simulation is executed again until a negative robustness value is obtained or until the user-defined maximum number of simulation executions is reached. In this case study, we utilize the Simulated Annealing optimization method to search for parameters that minimize the robustness value.

*Autonomy Monitors:* The autonomy monitoring service is implemented inside UxAS to monitor the positions of vehicles over time and decide if a task has completed or failed. Furthermore, it computes the robustness value of the trajectory with respect to the task requirements. First, we define specific monitors for each type of task and operating region constraints in our overall mission

specification. The monitors receive periodic timestamped messages containing the positions and headings of the various airplanes. It then updates the current completion status for each task and operating region constraints. It then publishes success or failure messages along with robustness values to S-Taliro.

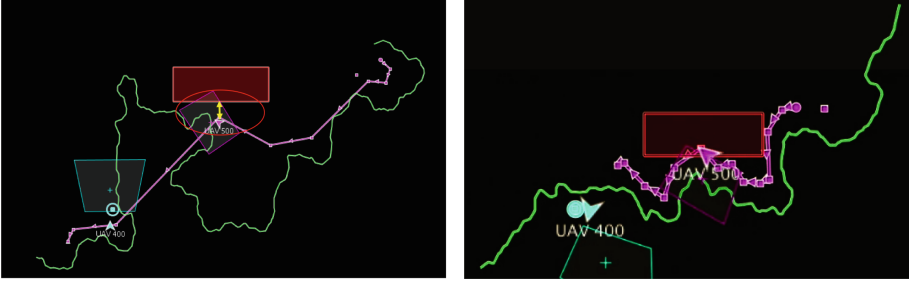
We now briefly describe the operation of S-Taliro to generate test cases. This includes randomized generation of paths for various search tasks and the random generation of operation zone constraints.

*Random Path Generation:* A random path is specified by its starting and ending points:  $p_{start}, p_{end}$ , minimum and maximum distances between various segments of the path  $d_{min}, d_{max}$ , maximum angular difference between segments  $\theta_{max}$ , and the standard deviation of the angle difference,  $\theta_{\sigma}$ . We generate a list of coordinates  $p_0 : p_{start}, \dots, p_N : p_{end}$  with the line joining the coordinates specifying the overall path.

Initially, the partial path just consists of  $p_{start}$ . We then sample a point at a sample distance  $d_{sample}$  and angle  $\theta_{sample}$ .  $d_{sample}$  is chosen randomly from the given range, and  $\theta_{sample}$  is chosen at random with the mean value centered around the line joining the last point to the end point  $p_{end}$  and specified standard deviation. This process continues until the last point in the path so far is close enough to the endpoint. At this stage, the point  $p_{end}$  is added to the list and the process terminates. The green paths in Fig. 2 are generated by this algorithm for a path search task.

*Scenario Description:* Scenarios are described externally through an XML file that is read and transmitted to the UxAS system through a middleware layer. The message to the UxAS system contains mission information that includes aircraft descriptions and configurations, initial aircraft states, target tasks, Keep In/Out zone parameters and weather conditions which can change during the simulation. In our approach, we mainly generate new test cases by modifying the parameters defined in these messages. We add additional fields to the xml structure to define the ranges of the parameters.

*Random Placement of Keep Out Zone:* A keep out zone specifies a region that an aircraft cannot enter during the mission. To test the system, we randomly place Keep Out zones ensuring that the overall mission remains feasible in doing so: i.e., no keep out zone intersects a path or target point in the mission specification. Our approach starts by computing the interval hull of the generated path and sampling a point in the hull. We then place the keep out region  $R$  at this point and test for an intersection with the path. If an intersection occurs, we then find a point of intersection and choose a direction along with we translate the region  $R$  by the minimum possible distance to move the current intersection point outside the region  $R$ . We repeat this process until we are free of intersections. However, this process need not terminate in all cases. To aid termination, we place maximum limits on the number of iterations and restart afresh. Examples of randomly generated paths (in green) with a randomly placed keep out zones (in red) are shown in Fig. 2.



**Fig. 2.** Randomly generated search paths and Keep Out zones along with simulations: (left) satisfying requirements and (right) violation. (Color figure online)

### 3.1 Case-Study

The scenario in our case study involves three UAVs with ID 400, 500 and 600, for which we denote the positions by  $p_{V400}, p_{V500}, p_{V600}$ . A line search task arrives, and UxAS generates an optimal plan for one of the UAVs to perform the task of obtaining surveillance video that covers the path to be searched. We also add a keep out zone  $Z_1$  at random such that it does not intersect with the search path. Since the aircrafts are not supposed to fly over keep out zones, we require that “Whenever any of UAV 400, 500 or 600 enters the keep out zone  $Z_1$ , it should exit the zone in 10s”. The MTL representation for this requirement can be given as

$$\square(r_1 \implies \diamond_{[0,10]}\neg r_1) \wedge \square(r_2 \implies \diamond_{[0,10]}\neg r_2) \wedge \square(r_3 \implies \diamond_{[0,10]}\neg r_3),$$

where  $r_1$  is a predicate which evaluates to True ( $\top$ ) when the UAV 400 enters the keep out zone  $Z_1$ , i.e.,  $r_1 := p_{V400} \in Z_1$ . Similarly,  $r_2 := p_{V500} \in Z_1$  and  $r_3 := p_{V600} \in Z_1$ .

We leave the shape of the path to be searched, the nominal speed values for each of the aircrafts, wind speed and wind directions that can change 6 times over the simulation time as the variable parameters of the scenario. We use OpenAMASE to simulate the aircrafts. In our test generation approach, we randomly sample a path as the line search task and use S-TALIRO to search over the variable scenario parameters to discover the system behaviors that do not satisfy the MTL specification which is given above. The system under test can be considered as the UxAS tool together with the UAVs in the scenario.

The aircrafts start from their given initial positions. If an aircraft never enters zone  $Z_1$  during the simulation, then the robustness value (for this example) is the minimum distance between the aircraft trajectory and the keep out zone boundaries. If an aircraft actually enters zone  $Z_1$ , then the robustness value for the violation reduces to how far inside  $Z_1$  the UAV flies beyond the 10 second time limit. Moreover, in the latter case, the robustness value is negative indicating that the requirement has been violated.

In the execution example given in Fig. 2 (left) the green path is the one to be searched and the purple path connects waypoints generated by UxAS. In this

case, the robustness value is the length represented by the yellow arrow which is the point where the aircraft comes closest to the keep out zone, as illustrated by the yellow arrow.

In our case study, S-TALiRO discovered cases where UxAS generates way-points inside a keep out zone for a path search task and the vehicles fly into this zone (see Fig. 2 (right)) and stay inside the zone for more than 10 seconds which is against the system requirements.

## 4 Conclusion and Future Work

We have developed a framework which can be used to automatically generate test cases which can discover scenarios that lead to unexpected behaviors. Although the level of automation can be increased, we have automated most of the process by extracting data from existing scenario files. This ability would save human effort spent on generating test cases. Furthermore, because the automatic test generation framework does not have a developer's/tester's bias, it can discover unforeseen conditions leading to failure.

As a future work, we propose to use Simulink aircraft simulation models to apply coverage guided test generation techniques. Furthermore, we plan to utilize simplified system dynamics for the aircrafts and the environment to compute functional gradient descent directions as described in our earlier work [11].

## References

1. Abbas, H., Fainekos, G., Sankaranarayanan, S., Ivančić, F., Gupta, A.: Probabilistic temporal logic falsification of cyber-physical systems. *ACM Trans. Embed. Comput. Syst. (TECS)* **12**(2s), 95 (2013)
2. Air Force Research Laboratory, Aerospace System Directorate, Power and Control Division. OpenAMASE, Aerospace Multi-agent Simulation Environment, December 2017. <https://github.com/afrl-rq/OpenAMASE/>
3. Air Force Research Laboratory, Aerospace System Directorate, Power and Control Division. OpenUXAS, Project for multi-UAV cooperative decision making, Dec. 2017. Available at <https://github.com/afrl-rq/OpenUxAS>
4. Annpureddy, Y., Liu, C., Fainekos, G., Sankaranarayanan, S.: S-TALiRO: a tool for temporal logic falsification for hybrid systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 254–257. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-19835-9\\_21](https://doi.org/10.1007/978-3-642-19835-9_21)
5. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. *Theoret. Comput. Sci.* **410**(42), 4262–4291 (2009)
6. Kingston, D., Rasmussen, S., Humphrey, L.: Automated UAV tasks for search and surveillance. In: 2016 IEEE Conference on Control Applications (CCA), pp. 1–8. IEEE (2016)
7. Koymans, R.: Specifying real-time properties with metric temporal logic. *Real-Time Syst.* **2**(4), 255–299 (1990)
8. Nghiem, T., Sankaranarayanan, S., Fainekos, G., Ivančić, F., Gupta, A., Pappas, G.J.: Monte-Carlo techniques for falsification of temporal properties of non-linear hybrid systems. In: HSCC 2010, pp. 211–220. ACM, New York (2010)

9. S-TaLiRo. December 2017. <https://sites.google.com/a/asu.edu/s-taliro/s-taliro>
10. Sutton, M., Greene, A., Amini, P.: Fuzzing: Brute Force Vulnerability Discovery. Pearson Education, London (2007)
11. Tuncali, C.E., Yaghoubi, S., Pavlic, T.P., Fainekos, G.: Functional gradient descent optimization for automatic test case generation for vehicle controllers. In: 2017 IEEE International Conference on Automation Science and Engineering (CASE). IEEE (2017)



# MoDeS3: Model-Based Demonstrator for Smart and Safe Cyber-Physical Systems

András Vörös<sup>1,2(✉)</sup>, Márton Búr<sup>1,4</sup>, István Ráth<sup>2,3</sup>, Ákos Horváth<sup>2,3</sup>,  
Zoltán Micskei<sup>2</sup>, László Balogh<sup>2</sup>, Bálint Hegyi<sup>2</sup>, Benedek Horváth<sup>2</sup>,  
Zsolt Mázló<sup>2,3</sup>, and Dániel Varró<sup>1,2,4</sup>

<sup>1</sup> MTA-BME Lendület Cyber-Physical Systems Research Group,  
Budapest, Hungary  
{vori,bur}@mit.bme.hu

<sup>2</sup> Department of Measurement and Information Systems,  
Budapest University of Technology and Economics, Budapest, Hungary  
rath@incquerylabs.com, {ahorvath,micskei,varro}@mit.bme.hu

<sup>3</sup> IncQuery Labs Ltd., Budapest, Hungary

<sup>4</sup> Department of Electrical and Computer Engineering,  
McGill University, Montreal, Canada

**Abstract.** We present MoDeS3, a complex research demonstrator illustrating the combined use of model-driven development, formal verification, safety engineering and IoT technologies for smart and safe cyber-physical systems. MoDeS3 represents a smart transportation system-of-systems composed of a model railway and a crane which may automatically load and unload cargo from trains where both subsystems need to fulfill functional and safety requirements. The demonstrator is built by using the model-based software engineering principle, while the system level safety is ensured by the combined use of design-time and runtime verification and validation techniques.

**Keywords:** Smart cyber-physical systems  
Model-driven engineering · Formal methods · Education  
Demonstrator

## 1 Introduction

**Motivation.** A smart and safe cyber-physical system (CPS) autonomously perceives its operational context and adapts to changes over an open, heterogeneous and distributed platform with a massive number of nodes, dynamically acquires available resources and aggregates services to make real-time decisions, and resiliently provides critical services in a trustworthy way [9, 12].

These challenges and the multidisciplinary nature of CPS make the engineering of such systems very complex. On the one hand, traditional techniques used for developing safety-critical systems may have limited applicability for CPS



[8]. Moreover, both research and education of CPSs necessitate well-documented open-source demonstrator platforms which capture and reflect the essence of problems and challenges, yet it is reasonably complex to highlight the key characteristics of CPSs and present them in the context of modern technologies.

**Objectives.** We introduce *MoDeS3: the Model-based Demonstrator for Smart and Safe Cyber-Physical Systems*<sup>1</sup>, which aims to illustrate the combined use of model-driven development, intelligent data processing, safety engineering and IoT technologies in the context of safety-critical system of systems with emerging safety hazards. This open source project simultaneously serves as (1) a *research platform used for experimental evaluation* of CPS-related research, (2) a *complex educational platform* used for graduate and undergraduate teaching, and (3) an *IoT technology demonstrator* used by industrial partners and collaborators.

**The MoDeS3 demonstrator as a smart and safe CPS.** The physical layout of MoDeS3 is depicted in Fig. 1. As its core is a *model railway transportation system*, guarantees for the safe operation of trains, switches, and semaphores are required. Connected to a specific segment of the track, an automated *crane system* loads cargo on and off the trains. As such, it is a critical system in itself since the cargo cannot be dropped by the crane.

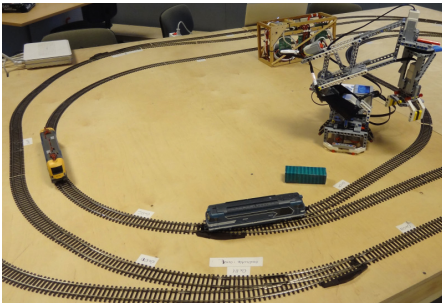


Fig. 1. Physical layout

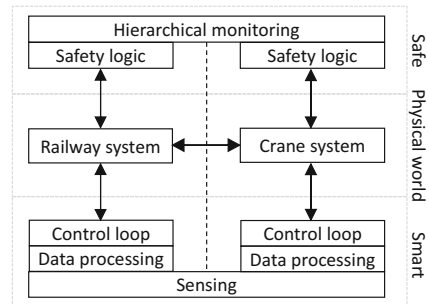


Fig. 2. Architectural overview

Additionally, the MoDeS3 demonstrator represents a system-of-systems, since the railway and the crane system are physically located next to each other. In this case, new kind of hazardous situations may emerge which are not incorporated in any of the constituent systems. For instance, a rotating movement of the crane may physically hit a train passing by along the track.

To make the demonstrator more realistic, we adopted various safety assurance techniques ranging across design-time formal verification and validation (V&V), runtime monitoring or testing on various levels of abstraction (see Sect. 2). A conceptual overview is provided in Fig. 2. Multiple levels of safety are applied: a distributed safety logic is responsible for the accident-free operation of the trains. Hierarchical monitoring is used to ensure the safe cooperation of the

<sup>1</sup> <http://modes3.inf.mit.bme.hu/>.

subsystems. The details are given in Sect. 2. A wide range of sensors serves as a rich information source for smart control and data analytics (see details in Sect. 3). Educational use of MoDeS3 is covered in Sect. 4. The project timeline and conclusions are drawn in Sect. 5.

## 2 Design- and Runtime Assurance

The development of safety-critical systems has a long history with well-established methodologies to ensure safe operation. The MoDeS3 demonstrator was built using Model-based Systems Engineering (MBSE) where models are first-class citizens of the engineering process. SysML models are used to define the functional and the platform architecture of the system, while the Gamma Statechart Composition Framework<sup>2</sup> is used for the precise definition of the component level behaviour. Gamma supports the design, verification and code generation for component-based reactive systems.

The MoDeS3 demonstrator incorporates various V&V approaches (such as model checking, structural completeness and consistency analysis) as well as fault-tolerance techniques — all of which are widely used in real systems. However, due to its complex and multidisciplinary nature, design-time assurance cannot guarantee in itself the safe operation of inherently dynamic smart CPSs. Therefore, runtime certification [13] using techniques like runtime monitoring [10] or runtime verification [7] complement design-time assurance. Therefore, MoDeS3 integrates runtime monitoring and verification techniques on both component and system-level to flag violations of safety properties during the operation of the system and trigger appropriate counter-measures such as immediately stopping or slowing down trains. Our emphasis is on the combined use of design-time and runtime V&V techniques when building MoDeS3 to address its safety requirements. A high-level overview of V&V techniques is illustrated in Fig. 3.

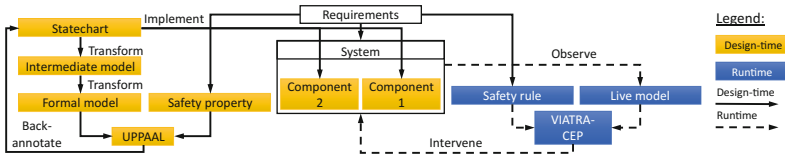


Fig. 3. Overview of design-time and runtime verification in MoDeS3

### 2.1 Design-Time Formal V&V of Timing Properties

As a primary design-time verification task, we carried out a formal analysis of logical and timing properties of the distributed safety logic of the accident prevention subsystem. We used the Gamma Statechart Composition Framework [11]

<sup>2</sup> <http://gamma.inf.mit.bme.hu/>.

to form the composite behavior of Yakindu statechart models. This composite model serves as the engineering input for the design-time analysis. Gamma introduces an intermediate state machine language with some high-level constructs and precisely defined semantics [14] to serve as a bridge between engineering and formal models. This intermediate language also helps in the back-annotation of analysis results to statechart models. Formal verification is performed using UPPAAL model checker [2], which is widely used for analyzing timing properties.

The generated formal models address the verification of a single component against local properties as well as their interaction against global properties. However, these models are insufficient to reason about the correctness of the system in themselves. For that purpose, one needs to ensure the interaction between the physical world and the cyber world.

For this purpose, formal models are built to capture the (logical and physical) behavior of trains. Then a combined design-time verification can reveal potentially unsafe situations, e.g. if trains move too fast, some accidents cannot be prevented. Investigating the counterexample retrieved by Gamma highlights that the situation could only happen if the trains are faster than the messages transmitted between the components. Unless there is a denial-of-service attack with flooding of messages, this is hardly the case in practice, but it is still a potential security threat. After extending the statechart models with timing assumptions on communication speed, we can formally prove that the safety logic prevents multiple trains from entering the same section of the track.

## 2.2 System-Level Runtime Monitoring

As smart and safe CPSs have complex interactions with an evolving environment and the physical world, we complement design-time verification in MoDeS3 with runtime monitoring techniques on both component and system level. For space considerations, here we only provide a summary of the *hierarchical system-level runtime monitoring* technique using graph reasoning with live models and complex event processing techniques (see right part of Fig. 3).

As traditional monitoring techniques consume events but do not cover data-dependent behavior or structural properties, runtime knowledge about the operational system is captured by a runtime (live) model [4]. A runtime model captures the current abstract snapshot of the system and its operational context, and changes in the underlying running system are constantly incorporated. Unlike a detailed design model, a runtime model only captures those aspects of the system, which are relevant for runtime monitoring and intervention.

System-level safety monitoring is carried out using graph queries and complex event processing (CEP) [5], which detect runtime violations of safety rules (by the identification of changes in the match sets of graph queries) and trigger appropriate reactions. While graph models and queries are widely used in *design tools* of CPS and CEP is a key technique in stream processing for web applications, their use in the context of smart and safe CPS is an innovative aspect of the MoDeS3 demonstrator.

Graph-based runtime techniques nicely complement traditional, component-level, automaton-based monitors deployed to embedded computers since critical signals raised by low-level monitors can be further propagated to the system-level as a hierarchy of events. As a consequence, we obtain a technique for the runtime monitoring of system-of-systems [15] where emerging and ad hoc hazardous situations can be incorporated and detected automatically also in the presence of complex structural (graph) constraints.

### 3 Smart IoT Technologies

Intelligent services and technologies are integrated into MoDeS3 at various levels. First, distributed autonomous intelligent control is used both for driving the trains and also to load and unload cargo on trains by the robot crane. Moreover, multiple sensors and surveillance cameras are used, and initial processing of the data stream is carried out close to the information source in accordance with fog and edge computing [6, 9] principles. Such sensor data can be consumed by multiple data processing services and different subsystems by offering generalized sensing services. This way, reusable smart sensing services may initiate actuation and control according to the collected environmental and operational information. The software stack is based on open-source Eclipse IoT solutions.

System-level runtime verification exploits events obtained from track sensors and general-purpose surveillance cameras. The visual information is processed using state-of-the-art computer vision (OpenCV) and neural network (TensorFlow) technologies. Distributed components are using state-of-the-art IoT communication protocols with open connectivity to share sensor data with different data processing services (and different subsystems). MQTT<sup>3</sup> provides a lightweight protocol for exchanging messages in a publish/subscribe model, which is widely used in communication between embedded devices and sensors.

Open-source microcontrollers (Arduino) and industrial embedded computers (Raspberry Pi, BeagleBone Black) provide the hardware elements of the platform. Cloud computing technologies are used for integrating hardware devices, service APIs and real-time data analytics.

### 4 MoDeS3 in Education

One of the goals of MoDeS3 is to support education with realistic examples and case studies. The demonstrator currently fulfills this purpose at the Budapest University of Technology and Economics at various stages of education.

**Undergraduate level.** At the first year introductory *System Modeling* course, the demonstrator is used for illustration purposes: students are introduced to modeling by the simplified models of the platform. Third year undergraduate students of the *Systems Engineering* course face the problem of designing the railway system by going through the development process. All phases of the

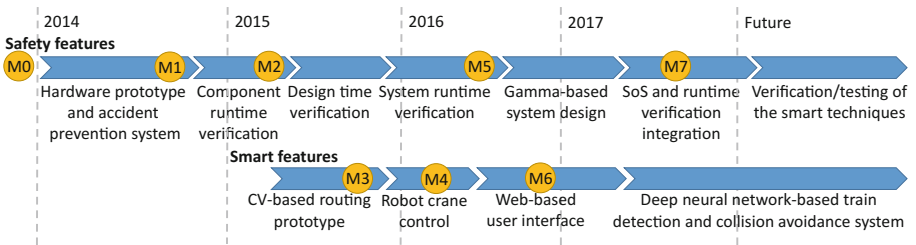
<sup>3</sup> <http://mqtt.org/>.

development process result in a model which is then evaluated by the instructors. Undergraduate students choose thesis project after completing the Systems Engineering course which may include developments of the MoDeS3 platform itself.

**Graduate level.** At the master’s level, three courses actively use the demonstrator platform. The course on *Model-Driven Software Development* introduces domain specific languages and development of model transformations for the students. The *Cyber-Physical Systems* course integrates the knowledge from the previous courses and introduces the modeling and controlling of hybrid systems. Beside the theoretical foundations, practical skills for integrating IoT technologies and cloud computing is also part of the curriculum. CPS course also covers fault-tolerance and other extra-functional aspects of cloud-based CPS. *Software and Systems Verification* is a course for further enhancing the knowledge of the students on testing with a specific focus on model-based testing or hardware-in-the-loop and model-in-the-loop testing. The course also summarizes runtime verification with a special focus on the hierarchical composition of the verification tasks according to the specification. At this part of the course, the advanced verification approaches are illustrated to the students on the MoDeS3 platform.

## 5 Project Timeline and Conclusion

Since its inception in 2014, the project has been proceeding by major milestones which have been organized along public demonstrations and presentations. At each milestone, some new features have been introduced, and critical maintenance tasks have been completed. These milestones are illustrated in Fig. 4 together with the new features.



**Fig. 4.** Project timeline and milestones. **M0:** Project kickoff, **M1:** Researchers’ Night 2014, **M2:** Ericsson University Day 2015, **M3:** Researchers’ Night 2015, **M4:** 2016 Eclipse IoT Challenge and Ericsson University Day 2016, **M5:** Researchers’ Night 2016 and EclipseCon France 2016, **M6:** EclipseCon Europe 2016 Demo, **M7:** EclipseCon Europe 2017 Demo

MoDeS3 demonstrates the innovative use of model-driven engineering approaches, formal methods and intelligent technologies for smart CPS. MoDeS3

proved its innovation at many industrial events: the team won a third prize at the Eclipse Open IoT Challenge 2.0 and MoDeS3 was exhibited twice at the industrial EclipseCon Europe conference and another workshop [1].

As a future work, we plan to further extend the demonstrator with smart technologies, such as a neural network based collision avoidance system and intelligent data analysis. Smart techniques used in for accident prevention have to be extensively tested/verified, where we will exploit the recent advances of the field. In addition, a novel distributed graph-based monitoring approach [3] will be integrated to provide an additional level of safety.

**Acknowledgment.** MoDeS3 is a joint effort of many participants. It was partially supported by MTA-BME Lendület Research Group on Cyber-Physical Systems the ARTEMIS JU R5-COP project and the NSERC RGPIN-04573-16 project. MoDeS3 also received financial and technical support from our industrial partners: IncQuery Labs Ltd., Quanopt Ltd., Ericsson Hungary and Miniversum. The TITAN Xp used for this research was donated by the NVIDIA Corporation. Colleagues at Dept. of Measurement and Information Systems (BME) worked on the project beside the authors: István Majzik, Gábor Szárnyas, and Oszkár Semeráth. We also thank the hard work of our students: Flórán Deé, Márton Elekes, Anna Gujgicz, Bence Graics, Raimund Konnerth, Gergő Somos, and Sámuel Várallyay.

## References

1. Balogh, L., et al.: Distributed and heterogeneous event-based monitoring in smart cyber-physical systems. In: MT CPS Workshop (CPS Week 2016) (2016)
2. Behrmann, G., et al.: UPPAAL 4.0. In: Third International Conference on the Quantitative Evaluation of Systems, pp. 125–126. IEEE (2006)
3. Búr, M., et al.: Distributed graph queries for runtime monitoring of cyber-physical systems. In: International Conference on Fundamental Approaches to Software Engineering (2018, accepted)
4. 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., Afmann, 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](https://doi.org/10.1007/978-3-319-08915-7_4)
5. Dávid, I., Ráth, I., Varró, D.: Foundations for streaming model transformations by complex event processing. *Softw. Syst. Model.* **17**(1), 1–28 (2016)
6. Dubey, A., et al.: Resilience at the edge in cyber-physical systems. In: FMEC, pp. 139–146, May 2017
7. Havelund, K.: Rule-based runtime verification revisited. *STTT* **17**(2), 143–170 (2015)
8. Lee, E.A.: Cyber physical systems: design challenges. In: 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing, pp. 363–369 (2008)
9. Lee, E.A., et al.: The swarm at the edge of the cloud. *IEEE Des. Test* **31**(3), 8–20 (2014)
10. Medhat, R., et al.: Runtime monitoring of cyber-physical systems under timing and memory constraints. *ACM T. Embed. Comput. Syst.* **14**(4), 1–29 (2015)
11. Molnár, V., et al.: The gamma statechart composition framework. In: ICSE 2018: Demonstrations (2018, accepted)

12. Nielsen, C.B., et al.: Systems of systems engineering: basic concepts, model-based techniques, and research directions. *ACM Comput. Surv.* **48**(2), 18 (2015)
13. Rushby, J.: Runtime certification. In: Leucker, M. (ed.) *RV 2008*. LNCS, vol. 5289, pp. 21–35. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-89247-2\\_2](https://doi.org/10.1007/978-3-540-89247-2_2)
14. Tóth, T., Vörös, A.: Verification of a real-time safety-critical protocol using a modelling language with formal data and behaviour semantics. In: Bondavalli, A., Ceccarelli, A., Ortmeier, F. (eds.) *SAFECOMP 2014*. LNCS, vol. 8696, pp. 207–218. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-10557-4\\_24](https://doi.org/10.1007/978-3-319-10557-4_24)
15. Vierhauser, M., et al.: Reminds: a flexible runtime monitoring framework for systems of systems. *J. Syst. Softw.* **112**, 123–136 (2016)

## Author Index

- Ahmad, Waqar 139  
Aït-Ameur, Yamine 1  
Aldous, Peter 367
- Balogh, László 460  
Baresi, Luciano 315  
Bendisposto, Jens 244  
Benyagoub, Sarah 1  
Bersani, Marcello M. 315  
Bhattacharyya, Siddhartha 20  
Bingham, Brad 95  
Blanchard, Allan 37  
Bochot, Thomas 399  
Broccia, Giovanna 54  
Búr, Márton 460
- Cabrera, Kalou 333  
Carvalho, Marco 20  
Cassel, Sofia 333  
Chand, Saksham 70  
Chaudhary, Kaylash 157, 164  
Chen, Charles Zhuo 87  
Colton, Simon 237
- David, Matthieu 399  
De Paula, Flavio M. 95  
Delmas, Rémi 399  
Dietl, Werner 87  
Ding, Guohui 452  
Dutertre, Bruno 113  
Dutta, Souradeep 121
- Elderhalli, Yassmeen 139  
Eskridge, Thomas C. 20
- Fainekos, Georgios 452  
Fehnker, Ansgar 157, 164  
Feliú, Marco A. 179  
Fromherz, Aymeric 185
- Gerhold, Marcus 203  
Giaquinta, Ruben 220  
Guiochet, Jérémie 333
- Haran, Arvind 95  
Hartmanns, Arnd 203  
Hasan, Osman 139  
Hegyí, Bálint 460  
Hoffmann, Ruth 220  
Horváth, Ákos 460  
Horváth, Benedek 460  
Hoxha, Bardh 452
- Ireland, Andrew 237  
Ireland, Murray 220
- Jha, Susmit 121  
Jovanović, Dejan 113
- Khurshid, Sarfraz 416  
Körner, Philipp 244  
Kosmatov, Nikolai 37
- Laarman, Alfons 261, 280  
Liu, Yanhong A. 70  
Llano, Maria Teresa 237  
Loulergue, Frédéric 37
- Maarand, Hendrik 299  
Marconi, Francesco 315  
Mashkoo, Atif 1  
Masson, Lola 333  
Mázló, Zsolt 460  
McCarthy, Jay 367  
Mehta, Vinay 164  
Meijer, Jeroen 349  
Mercer, Eric 367  
Micskei, Zoltán 460  
Milazzo, Paolo 54  
Miller, Alice 220  
Miné, Antoine 185  
Moscato, Mariano M. 179
- Nakade, Radha 367  
Narizzano, Massimo 383  
Navas, Jorge A. 113  
Neogi, Natasha A. 20  
Norman, Gethin 220



- Ochoa Escudero, César 399  
Ölveczky, Peter Csaba 54  
Oquadjaout, Abdelraouf 185  
Ouederni, Meriem 1
- Păsăreanu, Corina S. 416  
Pulina, Luca 383
- Qiu, Rui 416  
Quattrocchi, Giovanni 315
- Ráth, István 460  
Rocha, Camilo 435  
Romero, Miguel 435  
Rossi, Matteo 315
- Sankaranarayanan, Sriram 121, 452  
Stafford, Milton 20  
Stoelinga, Mariëlle 203
- Tacchella, Armando 383  
Tahar, Sofiène 139  
Tiwari, Ashish 121  
Törngren, Martin 333  
Tuncali, Cumhur Erkan 452
- Uustalu, Tarmo 299
- van de Pol, Jaco 349  
Varró, Dániel 460  
Vörös, András 460  
Vuotto, Simone 383
- Waeselynck, Hélène 333  
Wen, Junye 416  
Wiels, Virginie 399
- Yang, Guowei 416