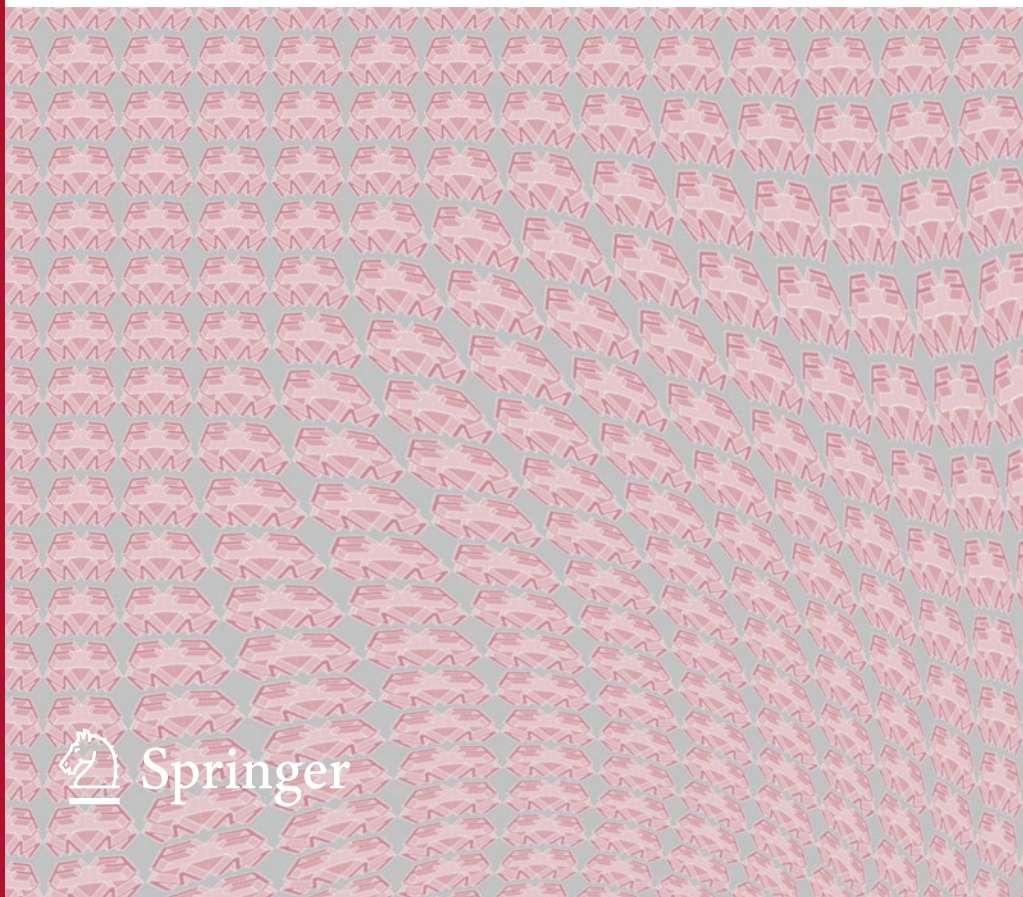
Formal Methods

LNCS 10886

Einar Broch Johnsen
Ina Schaefer (Eds.)

Software Engineering and Formal Methods

16th International Conference, SEFM 2018
Held as Part of STAF 2018
Toulouse, France, June 27–29, 2018, Proceedings



Springer

Lecture Notes in Computer Science **10886**

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

Formal Methods

Subline of Lectures Notes in Computer Science

More information about this series at http://www.springer.com/series/7407

Einar Broch Johnsen · Ina Schaefer (Eds.)

Software Engineering and Formal Methods

16th International Conference, SEFM 2018
Held as Part of STAF 2018
Toulouse, France, June 27–29, 2018
Proceedings

Springer

Editors
Einar Broch Johnsen
University of Oslo
Oslo
Norway

Ina Schaefer
Braunschweig University of Technology
Braunschweig
Germany

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-92969-9 ISBN 978-3-319-92970-5 (eBook)
https://doi.org/10.1007/978-3-319-92970-5

Library of Congress Control Number: 2018944412

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

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

Message from the STAF Organizers

Software Technologies: Applications and Foundations (STAF) is a federation of leading conferences on software technologies. It provides a loose umbrella organization with a Steering Committee that ensures continuity. The STAF federated event takes place annually. The participating conferences and workshops may vary from year to year, but they all focus on foundational and practical advances in software technology. The conferences address all aspects of software technology, from object-oriented design, testing, mathematical approaches to modeling and verification, transformation, model-driven engineering, aspect-oriented techniques, and tools. STAF was created in 2013 as a follow-up to the TOOLS conference series that played a key role in the deployment of object-oriented technologies. TOOLS was created in 1988 by Jean Bézivin and Bertrand Meyer and STAF 2018 can be considered its 30th birthday.

STAF 2018 took place in Toulouse, France, during June 25–29, 2018, and hosted: five conferences ECMFA 2018, ICGT 2018, ICMT 2018, SEFM 2018, TAP 2018, and the Transformation Tool Contest TTC 2018; eight workshops and associated events. STAF 2018 featured seven internationally renowned keynote speakers, welcomed participants from all around the world, and had the pleasure to host a talk by the founders of the TOOLS conference, Jean Bézivin and Bertrand Meyer.

The STAF 2018 Organizing Committee would like to thank (a) all participants for submitting papers and attending the event, (b) the Program Committees and Steering Committees of all the individual conferences and satellite events for their hard work, (c) the keynote speakers for their thoughtful, insightful, and inspiring talks, and (d) the Ecole Nationale Supérieure d'Electrotechnique, Electronique, Hydraulique et Télécommunications (ENSEEIHT), the Institut National Polytechnique de Toulouse (Toulouse INP), the Institut de Recherche en Informatique de Toulouse (IRIT), the région Occitanie, and all sponsors for their support. A special thanks goes to all the members of the Software and System Reliability Department of the IRIT Laboratory and the members of the INP-Act SAIC, dealing with all the foreseen and unforeseen work to prepare a memorable event.

Marc Pantel
Jean-Michel Bruel

Message from the SEFM Program Chairs

This volume contains the papers presented at SEFM 2018, the 16th International Conference on Software Engineering and Formal Methods, held June 27–29 in Toulouse, France. SEFM 2018 was collocated with STAF 2018.

The SEFM conference aims to bring together leading researchers and practitioners from academia, industry, and government, to advance the state of the art in formal methods, to facilitate their uptake in the software industry, and to encourage their integration within practical software engineering methods and tools. The topics of interest for submission included the following aspects of software engineering and formal methods:

- New frontiers in software architecture: self-adaptive, service-oriented, and cloud computing systems; component-based, object-based, and multi-agent systems; real-time, hybrid, and embedded systems; reconfigurable systems
- Software verification and testing: model checking and theorem proving; verification and validation; probabilistic verification and synthesis; testing
- Software development methods: requirement analysis, modeling, specification, and design; light-weight and scalable formal methods
- Application and technology transfer: case studies, best practices, and experience reports; tool integration
- Security and safety: security and mobility; safety-critical, fault-tolerant, and secure systems; software certification
- Design principles: domain-specific languages, type theory, abstraction, and refinement

SEFM 2018 solicited full research papers describing original research results, case studies and tools, and short papers on new ideas and work in progress, describing new approaches, techniques, and/or tools that are not fully validated yet. We received 58 submissions from 25 different countries. Each submission was reviewed by at least three Program Committee members. We accepted 17 regular papers, with an acceptance rate of 29%. The program also featured two remarkable invited talks:

- Mark Harman (Facebook and University College London): “We Need a Testability Transformation Semantics”
- Andrzej Wasowski (IT University of Copenhagen): “Hunting Resource Manipulation Bugs in Linux Kernel Code”

Our first words of thanks go to the Program Committee members and to the external reviewers, who carried out thorough and careful reviews and enabled the assembly of this high-quality work. We thank the authors for their submissions, and for their collaboration in further improving their papers. A special word of thanks goes to our invited speakers, Mark Harman and Andrzej Wasowski, for accepting our invitation and for their very stimulating contributions. We also thank the local Organizing

Committee of STAF 2018, who largely contributed to the success of this event. We also thank the developers and maintainers of the EasyChair conference management system, which was of great help in handling the paper submission, reviewing, discussion, and assembly of the proceedings. Finally, we are most grateful to Alexander Knüppel, who provided invaluable help in the preparation of the conference website and proceedings.

June 2018

Einar Broch Johnsen
Ina Schaefer

Organization

Steering Committee

Manfred Broy	Technische Universität, Munich, Germany
Radu Calinescu	University of York, UK
Antonio Cerone	Nazarbayev University, Kazakhstan
Alessandro Cimatti	FBK, Italy
Rocco De Nicola	IMT School for Advanced Studies Lucca, Italy
Mike Hinchey	Lero, The Irish Software Engineering Research Centre, Ireland
Paddy Krishnan	Oracle Labs, Australia
Eva Khn	TU Wien, Austria
Zhiming Liu	Southwest University, China
Gwen Salan	Grenoble INP, Inria, LIG, France
Marjan Sirjani	Malardalen University, Sweden

Program Chairs

Einar Broch Johnsen	University of Oslo, Norway
Ina Schaefer	Technische Universität Braunschweig, Germany

Program Committee

Erika Abraham	RWTH Aachen University, Germany
Elvira Albert	Universidad Complutense de Madrid, Spain
Ade Azurat	Fasilkom UI, Indonesia
Luis Barbosa	University of Minho, Portugal
Dirk Beyer	LMU Munich, Germany
Marcello Bonsangue	Leiden University, The Netherlands
Jonathan Bowen	London South Bank University, UK
Mario Bravetti	University of Bologna, Italy
Einar Broch Johnsen	University of Oslo, Norway
Ana Cavalcanti	University of York, UK
Alessandro Cimatti	FBK-irst, Italy
Ferruccio Damiani	Università di Torino, Italy
Frank De Boer	CWI, The Netherlands
Rocco De Nicola	School for Advanced Studies Lucca, Italy
John Derrick	Unversity of Sheffield, UK
Anke Dittmar	University of Rostock, Germany
George Eleftherakis	The University of Sheffield, UK
José Luiz Fiadeiro	Royal Holloway, University of London, UK
Carlo A. Furia	Chalmers University of Technology, Sweden

Philipp Haller	KTH Royal Institute of Technology, Sweden
Klaus Havelund	Jet Propulsion Laboratory, USA
Rob Hierons	Brunel University, UK
Marieke Huisman	University of Twente, The Netherlands
Paddy Krishnan	Oracle, USA
Peter Gorm Larsen	Aarhus University, Denmark
Kung-Kiu Lau	The University of Manchester, UK
Martin Leucker	University of Lübeck, Germany
Tiziana Margaria	Lero, Ireland
Mercedes Merayo	Universidad Complutense de Madrid, Spain
Shin Nakajima	National Institute of Informatics, Japan
Viet Yen Nguyen	Hypefactors, Denmark
Fernando Orejas	Universitat Politècnica de Catalunya, Spain
Marc Pantel	IRIT/INPT, Université de Toulouse, France
Anna Philippou	University of Cyprus, Cyprus
Geguang Pu	East China Normal University, China
Leila Ribeiro	Universidade Federal do Rio Grande do Sul, Brazil
Philipp Ruemmer	Uppsala University, Sweden
Bernhard Rumpe	RWTH Aachen University, Germany
Gwen Salaün	University of Grenoble Alpes, France
Augusto Sampaio	Federal university of Pernambuco, Brazil
Cesar Sanchez	IMDEA Software Institute
Ina Schaefer	Technische Universität Braunschweig, Germany
Neeraj Singh	INPT-ENSEEIHT/IRIT, University of Toulouse, France
Marjan Sirjani	Malardalen University, Sweden; Reykjavik University, Iceland
Graeme Smith	The University of Queensland, Australia
Bernhard Steffen	Technische Universität Dortmund, Germany
Maurice H. Ter Beek	ISTI-CNR, Pisa, Italy
Willem Visser	Stellenbosch University, South Africa
Bruce Watson	Stellenbosch University, South Africa
Heike Wehrheim	University of Paderborn, Germany
Wang Yi	Uppsala University, Sweden
Ingrid Chieh Yu	University of Oslo, Norway

Additional Reviewers

Abd Alrahman, Yehia	Din, Crystal Chang
Arshad, Rehman	Dokter, Kasper
Barbon, Gianluca	Foster, Simon
Cardone, Felice	Freitas, Fred
Convent, Lukas	Galletta, Letterio
Correas Fernández, Jesús	García, Miriam
Dangl, Matthias	Giraudo, Mauro

Gordillo, Pablo
Gossen, Frederik
Inverso, Omar
Irfan, Ahmed
Isabel, Miguel
Jafari, Ali
Krishna, Ajay
Kulik, Tomas
Laarman, Alfons
Lamprecht, Anna-Lena
Lange, Felix Dino
Lemberger, Thomas
Lu, Yi
Macedo, Hugo Daniel
Markin, Grigory
Mauro, Jacopo
Micheli, Andrea
Mohaqeqi, Morteza
Nellen, Johanna
Netz, Lukas
Neves, Renato
Paolini, Luca
Pauck, Felix
Peng, Cong
Pérez, Jorge A.
Qian, Chen
Raco, Deni
Rüthing, Oliver
Sabouri, Hamideh
Sacerdoti Coen, Claudio
Santini, Francesco
Schupp, Stefan
Sharma, Arnab
Spagnolo, Giorgio O.
Stolz, Volker
Ta, Quang-Trung
Tapia Tarifa, Silvia Lizeth
Thoma, Daniel
Thomsen, Michael Kirkedal
Varga, Simon
von Wenckstern, Michael
Winter, Kirsten
Zhang, Ning

Contents

Invited Keynote

We Need a Testability Transformation Semantics 3
Mark Harman

Specification

From Software Specifications to Constraint Programming 21
Stefan Hallerstede, Miran Hasanagić, Sebastian Krings, Peter Gorm Larsen, and Michael Leuschel

Automated Specification Extraction and Analysis with Specstractor 37
Christoph Schulze, Rance Cleaveland, and Mikael Lindvall

Bridging the Gap Between Informal Requirements and Formal Specifications Using Model Federation 54
Fahad Rafique Golra, Fabien Dagnat, Jeanine Souquières, Imen Sayar, and Sylvain Guerin

Concurrency

Program Verification for Exception Handling on Active Objects Using Futures 73
Crystal Chang Din, Rudolf Schlatte, and Tzu-Chun Chen

Spread the Work: Multi-threaded Safety Analysis for Hybrid Systems 89
Stefan Schupp and Erika Ábrahám

FastLane Is Opaque – a Case Study in Mechanized Proofs of Opacity 105
Gerhard Schellhorn, Monika Wedel, Oleg Travkin, Jürgen König, and Heike Wehrheim

Program Analysis

Monte Carlo Tree Search for Finding Costly Paths in Programs 123
Kasper Luckow, Corina S. Păsăreanu, and Willem Visser

A Cloud-Based Execution Framework for Program Analysis 139
Daniel Balasubramanian, Dmitriy Kostyuchenko, Kasper Luckow, Rody Kersten, and Gabor Karsai

Cross-Architecture Lifter Synthesis 155
Rijnard van Tonder and Claire Le Goues

Model Checking and Runtime Verification

Counterexample Simplification for Liveness Property Violation 173
Gianluca Barbon, Vincent Leroy, and Gwen Salaün

Online Enumeration of All Minimal Inductive Validity Cores 189
Jaroslav Bendík, Elaheh Ghassabani, Michael Whalen, and Ivana Černá

Prevent: A Predictive Run-Time Verification Framework Using Statistical Learning 205
Reza Babaee, Arie Gurfinkel, and Sebastian Fischmeister

Applications

Formal Verification of Platoon Control Strategies 223
Adnan Rashid, Umair Siddique, and Osman Hasan

Exploring Properties of a Telecommunication Protocol with Message Delay Using Interactive Theorem Prover 239
Catherine Dubois, Olga Grinchtein, Justin Pearson, and Mats Carlsson

Automated Validation of IoT Device Control Programs Through Domain-Specific Model Generation 254
Yunja Choi

Shape Analysis and Reuse

Graph-Based Shape Analysis Beyond Context-Freeness 271
Hannah Arndt, Christina Jansen, Christoph Matheja, and Thomas Noll

Facilitating Component Reusability in Embedded Systems with GPUs 287
Gabriel Campeanu

Author Index 303

Invited Keynote

We Need a Testability Transformation Semantics

Mark Harman[1,2(✉)]

[1] Facebook, London, UK
mark.harman@ucl.ac.uk
[2] University College, London, UK

Abstract. This paper (This paper is a brief outline of some of the content of the keynote by the author at the 16^{th} International Conference on Software Engineering and Formal Methods (SEFM 2018) in Toulouse, France; 27th–29th June 2018.) briefly reviews Testability Transformation, its formal definition, and the open problem of constructing a set of formal test adequacy semantics to underpin the current practice of deploying transformations to help testing and verification activities.

1 Introduction

Testability transformation modifies a program to make it easier to test. Unlike traditional program transformation [18,43], which alters a program's syntax without changing its input–output behaviour, Testability Transformation may alter functionality. Nevertheless, it does respect a program semantics, defined by the test adequacy criterion. Therefore, the sense in which a Testability Transformation is meaning preserving rests on a formal definition of the semantics of transformations that preserve test adequacy. Sadly, to date, such a test adequacy semantics has yet to be formally defined.

There are several widely-used test adequacy criteria in practice, such as statement, branch, data flow, path and mutation adequacy [8,9,31]. Each of these adequacy criteria gives rise to a different test adequacy semantics, the definitions of which and their relationship as a formal lattice of semantics remain interesting and important open scientific problems at the intersection of Software Engineering and Formal Methods (SEFM). Tackling this set of related semantic definitions will provide a firm mathematical foundation for Testability Transformation, and by extension, may also yield insights into currently-deployed testing techniques. This paper aims to draw out some of these open problems as research questions for the SEFM research community.

Testability Transformation itself, was first formalised in 2004 [25]. However, informally, software engineers have performed transformations to aid testability for considerably longer. For example, it is routine for testers to 'mock up' procedures to allow testing of the 'whole' before the 'parts' have been fully implemented. Similarly, for verification purposes, it is often necessary to model parts of the system with stubs, where source code is unavailable for analysis [2].

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 3–17, 2018.
https://doi.org/10.1007/978-3-319-92970-5_1

These simple mock–and–model transformations are relatively well-understood. By contrast, other transformations, that have the potential to more dramatically benefit the effectiveness and efficiency of testing and verification, remain less well-understood, because of a lack of proper formal underpinnings. These more ambitious Testability Transformations exploit the way in which transformation rules can be 'more aggressive' in their disruption of the input–output behavior, so long as they take care to preserve test adequacy semantics.

For example, Testability Transformation has been used to tackle problems such as flag variables [4,5,20], unstructured control flow [28], data flow testing [34], state-based testing [33], and nesting [40]. An overview of these applications of Testability Transformation can be found in the 2008 survey [23]. As well as source code transformation, it has also been applied, more recently at bytecode level [37]. It has also been applied to alleviate problems in dynamic symbolic execution [11] and to tackle the oracle problem in testing [39].

Nevertheless, the lack of formal underpinning means that much of this work rests on, as yet, uncertain foundations and this inhibits progress; transformations should take care to preserve test adequacy semantics, *yet there is no formal description* of test adequacy semantics for any of the widely-used adequacy criteria. Much more could undoubtedly be achieved in terms of practical transformation benefit, if only the designers of Testability Transformations had a formal test adequacy semantics to which they could appeal. Practitioners and researchers could use such a semantics to motivate, justify or otherwise explore their transformation spaces.

2 Formal Definition of Testability Transformation

The goal of a Testability Transformation is to make it easier to test the untransformed program. Although test data is ultimately applied to the original program, the technique that generates it uses the transformed version because it is easier. The transformation process therefore needs to guarantee that any adequate set of test data, generated and adequate for the transformed program, will also be adequate for the original program. This (test-based) meaning preservation guarantee replaces the more familiar guarantee, whereby traditional transformation preserves the input–output relationship of the untransformed program.

The formal definition of Testability Transformation is simple and has been known for some time [25]. In this section, we briefly review this formal definition of Testability Transformation, based on earlier work [22,25], which we augment by defining, slightly more formally, what is intended by the term 'test adequacy criterion':

Definition 1 (Test Adequacy Criterion). Let $\mathbf{P}$ be a set of programs, with an input space $\mathcal{I}$. A test adequacy criterion, c is a function from programs to predicates over sets of inputs:

$$c \in P \rightarrow 2^{\mathcal{I}} \rightarrow \{true, false\}$$

A test adequacy criterion determines whether a set of test inputs (i.e., test suite) meets the criterion of interest for the tester. An example is the branch adequacy criterion which is true, for test suite T and program P, when for every feasible branch, b in P, the test suite T contains at least one test that executes b. Another example is the statement adequacy criterion, which is true when every reachable statement is executed by the at least one test in the test suite.

In practice [3], since branch traversibility and statement reachability are both undecidable, testers settle for a measure of the percentage of branches (statements) covered, setting (rather arbitrary) percentage thresholds for satisfaction of the coverage predicate. Alternatively, practicing testers may choose to identify a set of important branches (statements) that are know to be feasible (and of interest) that would need to be covered in order for the test adequacy criterion to return true.

With the concept of tests adequacy criterion in hand, we can now define what it means to be a transformation that is concerned with both programs *and* the tester's chosen adequacy criterion, both of which might be transformed, as we shall see.

Definition 2 (Testing-Oriented Transformation). Let $\mathbf{P}$ be a set of programs and $\mathbf{C}$ be a set of test adequacy criteria. A program transformation is a partial function in $\mathbf{P} \rightarrow \mathbf{P}$. A *Testing-Oriented Transformation* is a partial function in $(\mathbf{P} \times \mathbf{C}) \rightarrow (\mathbf{P} \times \mathbf{C})$.

The test adequacy criterion, $\mathbf{C}$ refers to the overall criterion, which may be composed of a set of instances, each of which is denoted by lower case c. For instance, branch coverage is a possible choice for $\mathbf{C}$, while a particular instance, c, might capture the specific set of branches to be covered.

A testing-oriented transformation is defined to be a *partial* function, simply to allow that the transformation might be undefined for some programs; the possibility that a testing-oriented transformation may fail to terminate is not ruled out. In practice, this is relatively unimportant generalisation from total functions, because it will always be acceptable to leave the program untransformed. Therefore, a partial testing-oriented transformation can be easily converted to a total testing-oriented transformation.

Definition 3 (Testability Transformation). A Testing-Oriented Transformation, τ is a *Testability Transformation* iff, for all programs p, criteria c, and test sets T, T is adequate for p according to c if T is adequate for p' according to c', where $\tau(p, c) = (p', c')$.

In some cases, the test adequacy criterion can be transformed along with the program under test and this definition allows for that. We call this a co-transformation (that is, one in which both the program under test and the test adequacy criteria are transformed). For instance, MC/DC test adequacy [49] can be safely co-transformed to branch adequacy provided that the program is co-transformed to expand the boolean logic operator terms in predicates [8]. This co-transformation means that branch coverage of the expanded program

is equivalent (has the same test adequate test input sets) as MC/DC on the untransformed version.

In other circumstances, it will be more convenient to leave the adequacy criterion untransformed, and thereby to produce a program transformation that respects it. That is, for some criterion, c, a c–preserving Testability Transformation guarantees that the transformed program is suitable for testing with respect to the original criterion; a widely-used scenario in practice.

Definition 4 (c–Preserving Testability Transformation). Let τ be a Testability Transformation. If, for some criterion, c, for all programs p, there exists some program p' such that $\tau(p, c) = (p', c)$, then τ is called a c–preserving Testability Transformation.

As an illustrative example, consider the program

Example 1. *Simple program under test illustration*

```
input(z);
x=1; y=z;
if (y>3) x=x+1;
    else x=x-1;
output(x,y,z)
```

Example 1 can be transformed to

```
input(z)
if (z>3) ;
    else ;
output(x,y,z)
```

Such a transformation does not preserve the effect of the original program on the variables `x` and `y` and, therefore, does not preserve the input–output behaviour of Example 1. However, it does preserve the set of sets of inputs that execute (or 'cover' in testing nomenclature) all branches. It also preserves the set of sets of inputs that cover all statements, despite the fact that the transformation process has removed some statements. Therefore, the transformation can be said to be a 'branch–adequate transformation'. It is also a 'statement–adequate transformation'.

This guarantee of test-adequacy for the transformation has practical significance: test inputs can be constructed from the transformed program, safe in the knowledge that any set of inputs that is branch adequate for the transformed program will also be branch adequate for the original (Example 1). Since branch adequacy subsumes statement adequacy, we would also immediately know that such a set of test inputs would also be statement adequate for Example 1.

Of course, when executed on each version, the input–output relationship will be different, but this is unimportant for test input generation; the test inputs, once generated, will be applied to Example 1, *not* the transformed version.

The advantage of the transformation is that it may prove to be more effective and/or efficient to generate tests from the transformed program than the original. This has been the motivation for the previous work on Testability Transformation, which has revealed many such cases whereby transformation eases test input generation and verification.

Testability transformation is not only useful, but also offers interesting intellectual and scientific challenges: transformations can cover different paths in the transformed program, yet still preserve path adequacy, because the sets of test inputs that cover all paths in the transformed program also cover all the paths in the original. The behaviour of transformed programs and their relationship to the original from which they are constructed are thus highly subtle and a full understanding clearly necessitates a proper formal semantic treatment. As a simple illustration of this need, consider the simple fragment F, defined below:

Example 2. (F)

```
input(x);
if (x>0) x=-1;
    else x=1;
output(x)
```

The fragment F can be branch-adequately transformed to F' defined below:

Example 3. (F')

```
input(x);
if (x>0);
    else;
output(x)
```

This transformation is also path-adequate. Furthermore, $F \circ F$ can be path-adequately transformed to $F' \circ F'$. That is, as illustrated in Fig. 1, test suites that cover all branches of $F' \circ F'$, also cover all branches of $F \circ F$, even though the test inputs will traverse different branches. Indeed, test inputs that cover all paths in $F' \circ F'$ *also* cover all paths in $F \circ F$, even though they follow *different* paths in the transformed program than they follow in the original.

Formal treatments of slicing may provide a starting point for understanding this behaviour [7,13,42,46], since we are partially concerned with the interplay between data and control flow. Slicing makes a good starting point for understanding a subclass of Testability Transformations for two reasons:

1. Slicing reduces program size and is, thereby, likely to reduce execution time, which may, in turn, help improve the performance of test techniques [24]. Slicing will likely be particularly useful for test techniques that require repeated execution of the program under test to generate test inputs [12,26], because the slice computation cost can be offset by the multiple test execution time gain.

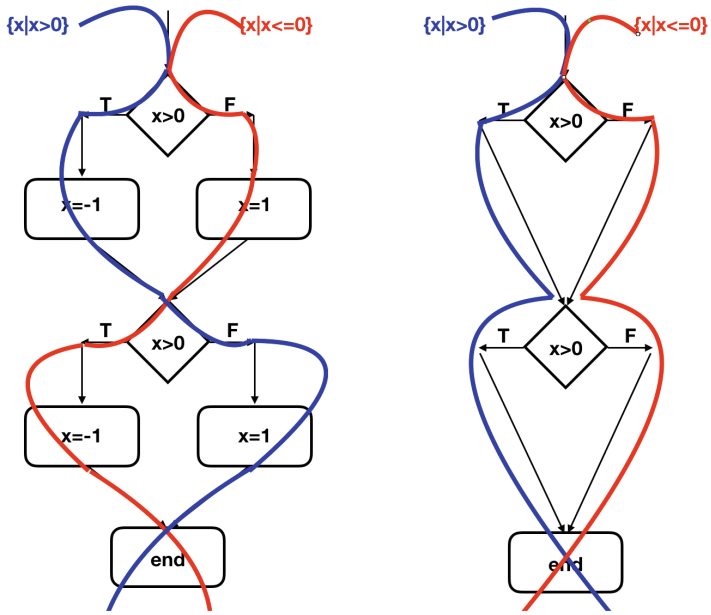


Fig. 1. The version on the right is both a branch-adequate and a path-adequate transformation of the version on the left, even though all possible test inputs follow different paths in each program. The original program partitions the set of all inputs (all initial values of `x`) into two disjoint subsets, each of which causes execution to follow one of two paths. Two test inputs, one from each subset, are necessary and sufficient to cover all (feasible) paths. The transformed program creates the same test input partition and so it preserves path-adequacy. This is illustrated by the input sets located at the top of each figure, and the path that inputs drawn from each follows. These two paths for each program are depicted in red and blue (or light and dark grey when printed in black-and-white). Observe that the feasible path sets are different for the two programs, and these two disjoint input subsets therefore follow different paths in each program. However, all input sets that cover (follow) all feasible paths in the transformed program on the right, nevertheless, also follow all feasible paths in the original on the left; the transformation is path adequacy preserving.

2. Slicing on all predicates for which we seek branch adequacy will maintain the computation needed for these predicates to correctly replicate their behaviour in the unsliced original program, thereby ensuring sufficiency of branch adequacy for these predicates.

However, although it may sufficient for path-adequacy, preserving control flow does not appear to be *necessary*, as illustrated in Fig. 1.

3 Testability Transformation and Abstract Interpretation

There exist valid Testability Transformations for which the allowable transformations are neither subsets nor supersets the traditional transformation set

that preserves the input–output behaviour of the untransformed program. For instance, consider this simply illustrative example:

Example 4 (More Concrete). `if (x>y) ; else ;`

This program can be transformed to the empty program, while preserving the input–output behaviour of untransformed program. However, this apparently 'simple optimization' does not preserve the sets of test suites that cover all branches of the original and, therefore, it cannot be a branch–adequate transformation. Therefore, there exist Testability Transformation semantics that are clearly not simply more abstract (allow a superset of transformations) than the conventional input–output relation semantics.

Furthermore, consider the program:

Example 5 (More Abstract). `if (x>y) x=1; else x=2;`

This program can be branch–adequately transformed to:

```
if (x>y) ; else ;
```

Clearly, such a branch–adequate transformation does not preserve the conventional input–output semantics. Therefore, branch–adequacy semantics is also not more concrete than conventional semantics.

From Examples 4 and 5 we are forced to conclude that branch–adequacy semantics is neither universally more concrete nor more abstract than conventional input–output relation semantics. Similar arguments can be made for other test adequacy criteria and the semantics they denote.

However, this observation does not mean that the consideration of the abstraction level of the semantics preserved has no role to play in formalising Testability Transformation. Quite the opposite: it should be possible to construct a lattice of Testability Transformation semantics ordered by semantic abstraction [22]. Therefore, abstract interpretation [16] would also be a promising framework within which to explore test adequacy semantics.

Perhaps a trace-based semantics would be more suitable to capture the properties preserved by Testability Transformation. However, as the example in Fig. 1 demonstrates, the traces followed in two different programs can be entirely different and yet the two programs are, nevertheless, path adequate testability transformations of each other. This suggests that perhaps merely abstracting from simple traces may prove insufficient to capture the semantics of test adequacy[1].

[1] Author's note: I sincerely hope that I may be proved wrong in this conjecture, since a simple abstracted trace semantics that captures test adequacy (and thereby informs Testability Transformation) would shed much light on many testing problems.

4 Testability Transformation and Metamorphic Testing

Metamorphic testing [15] is one approach to tackle the test oracle problem [6]. The oracle problem is captured by the question:

"What output(s) should be expected for a given input?"

With metamorphic testing, the tester uses relationships between already observed test cases (input–output pairs) and properties of a as-yet-unseen outputs for some newly provided test input. In this way Metamorphic Testing offers an oracle, albeit an incomplete one, where either no oracle, or an even more incomplete oracle, was previously available. As a simple illustration we might have:

$$\text{if } p(i) = r \text{ then } p(f(i)) = g(r)$$

for some program p and so-called 'metamorphic relations' f and g.

In this way we define one test case $(f(i), g(r))$ in terms of another, (i, r), using metamorphic relations (f and g).

As a more concrete example, suppose our first test reveals that

$$\texttt{CheckBalance}(i) = r,$$

giving us output r, a balance in a bank account, for the input i, a bank account number. From this test case, we can generate a set of new tests, using a non-negative deposit, x:

$$\texttt{CheckBalance}(\texttt{Deposit}(i, x)) = r', \text{where } x \geq 0,$$

such that we expect the condition $r' \geq r$. That is, we do not know the correct value r' should take (because we lack a complete test oracle). Nevertheless, we do have a *property* that r' should satisfy, for this new input x, expressed in terms of some previously witnessed test case (i, r).

A far more detailed and complete account of metamorphic testing can be found elsewhere [15]. For the present paper we merely wish to observe that metamorphic testing already lies at an intersection between Software Engineering and Formal Methods, because there is clearly a need to formalise the metamorphic relations for each metamorphic application area in order to provide a sound foundation for each.

It is also interesting to observe the close resemblance between metamorphic test relations and algebraic data type specifications [21]; both essentially capture a set of algebraic properties to be maintained in all valid instances. For instance, in the `CheckBalance` example above, metamorphic testing is simply exploiting the following algebraic property of `CheckBalance` and `Deposit` to find new test cases for old:

$$\begin{gathered} \forall x \in \mathbb{R}. \\ x \geq 0 \ \wedge \ \texttt{CheckBalance}(i) = r \ \wedge \ \texttt{CheckBalance}(\texttt{Deposit}(i, x)) = r' \\ \Rightarrow \\ r' \geq r \end{gathered}$$

which has a similar structure (and potential uses) as the familiar algebraic properties of data structures, such as:

$$\forall s \in \texttt{Stack}, x \in \texttt{Elem}.$$
$$\texttt{top}(\texttt{push}(x, s)) = x$$

There are at least two connections between Testability Transformation and Metamorphic Testing:

1. **Metamorphic Testing as Testability Transformation:** Consider again the metamorphic relations f and g for the program p, where

$$\text{if } p(i) = r \text{ then } p(f(i)) = g(r)$$

   If we re-write this, slightly, as

$$\text{if } p(i) = r \text{ then } (f \circ p)(i) = g(r)$$

   it becomes immediately clear that $f \circ p$ can be regarded as a (relatively simply) transformed version of p, and that the transformation of the oracle from $p(i) = r$ to $(f \circ p)(i) = g(r)$ can also be regarded as a transformation of the oracle, thereby resembling a test adequacy criterion transformation.
2. **Metamorphic Testability Transformation:** If we generalise a metamorphic equation like if $p(i) = r$ then $p(f(i)) = g(r)$, we can obtain a kind of 'Metamorphic Testability Transformation', which combines the metamorphic effect of the oracle with the transformation effect on the program. The generalisation would be if $p(i) = r$ then $(\mathcal{T}_{\text{p}}(p))(f(i)) = (\mathcal{T}_{\text{c}}(g))(r)$ where $\mathcal{T}_{\text{p}}$ is a Testability Transformation on programs and $\mathcal{T}_{\text{c}}$ is a Testability Transformation on the oracle; akin to a testability co-transformation of program and test adequacy criterion, as defined by Definition 3 of Testability Transformation.

These two possibilities for combining Metamorphic Testing and Testability Transformation are each, essentially, formalisations of McMinn's Co-testability Transformation, which transforms both the program under test and the corresponding oracle for that program under test [39].

5 Testability Transformation Research Questions to Be Tackled Using Formal Methods

In this section we give seven open problems concerning formal aspects of Testability Transformation, each of which could yield sufficient challenges for a connected set of research activities involving several non-trivial subprojects, perhaps suitable for a PhD programme or other grant-funded project at the intersection of Software Engineering and Formal Methods.

Research Question 1 (Novel Semantics). Can we define an elegant formal semantics for Branch Adequacy?

According to such a branch adequacy semantics, any meaning–preserving transformation would preserve the sets of sets of branch adequate inputs. Ideally, the semantics should admit specialisation to a specific set of branches, so there is also a need to formalise the specification of branches to support such a specialisation.

Branch adequacy is widely-studied and held out as a goal for many test techniques [10,45], making this good starting point. Progress on Research Question 1 alone could yield great progress in our understanding of many test input generation techniques.

There are several possible candidates for defining a test adequacy semantics. Different notations may be more valuable in different contexts. For instance, one might presume that algebraic semantics [19] would be intellectually close to the notions of transformation and might be best suited to declarative formulation and correctness proofs of transformation rules. By contrast, operational semantics [44] might best explain and inform the way in which test adequacy criteria have an operational character, being concerned with execution and, for several practically important criteria, execution paths. Finally, Hoare triples [1,29] might best capture the way that certain aspects of state must be maintained in order to preserve test behaviour, while others can be safely removed without affecting test adequacy.

Research Question 2 (Semantic Abstraction). What is the relationship between Testability Transformation and Abstract Interpretation?

As discussed in Sect. 3, Abstract Interpretation appears to be clearly relevant to Testability Transformation, but not to explain the relationship between test adequacy semantics and conventional semantics. Rather, it would appear that each of the different test adequacy criteria gives rise to a different semantics. It is the relationship between these different test adequacy semantics that may best be explained in terms of Abstract Interpretation.

An initial answer to Research Question 2 might start by considering the relationships between the test adequacy semantics for branch, statement and path coverage, all of which exhibit a *prima facie* structural relationship. It will also be interesting to compare the abstraction relationships between test adequacy criteria and the subsumption relationship between them; perhaps abstract interpretation can also provide a semantic framework for better understanding of test adequacy subsumption.

Research Question 3 (Mutation Semantics). What is the test adequacy semantics for mutation testing?

Mutation testing [31] has been shown to be highly effective as a test adequacy criterion [14,32]. Therefore, it would be interesting to formalise the semantics preserved by mutation testing. There are different forms of mutation, such as strong, firm and weak [48]; a formalisation would allow a more rigorous investigation of relationships between these different forms of mutation. It would also be useful to specialise a semantics, so that it can apply only to a specific set

of mutants, thereby facilitating a formal investigation of relationships between mutations sets and mutant subsumption [30,35].

Finally, since mutants are, themselves, transformations of the program under test, it would be fascinating to formalise the mutants themselves, and candidate fixes, within the same Testability Transformation formal semantic framework. This would facilitate a mathematical investigation of mutants as repair candidates [36], mutant equivalence [27,38,41], links between mutation and metamorphic testing [50], and of the empirically-observed phenomenon of mutational robustness [47].

Research Question 4 (Transformation Sets). What are the transformation rules that are correct for a given test adequacy criterion (branch, mutation...) and what proof obligations can be discharged when defining them?

Once some initial theoretical foundations have been laid, it would be highly useful for testing practitioners to have such sets of transformation rules. Formal semantic foundations imbue transformation sets with the safety-in-application that comes for formal guarantees of meaning preservation (with respect to a test adequacy criterion of interest to the tester). Work on abstract interpretation frameworks for specifying and reasoning about the correctness of transformation rules [17] may be useful here.

Research Question 5. What transformations can be performed on test adequacy criteria and what proof obligations do they raise?

Test adequacy criteria can be transformed, both with and without transforming the program under test. This has been known for some time (the example of MC/DC to branch coverage from Sect. 2 is one such example). However, we lack a formal framework within which to understand such criteria transformations with which we could attempt to prove their correctness.

Research Question 6. Can we define a formal semantics of Testability Transformation that helps us to better inform and understand the practice of Metamorphic Testing?

A formal understanding of this relationship would help to generalise both Testability Transformation and Metamorphic Testing, as set out in Sect. 4. Such a formalisation would also provide a new mathematical foundation for the study of metamorphic testing, using the formal semantics established in answer to Research Question 1. This would not only be theoretically helpful (yielding improved understanding and reasoning about Metamorphic Testing), it would also have practice benefit by extending the reach of Metamorphic Testing, to include transformed programs as well as transformed test oracles.

Research Question 7 (Anticipatory Testing). What Testability Transformations can be used to assist and pre-harden programs for Anticipatory Testing?

Anticipatory Testing is a (very recent) new approach to testing proposed by Tonella[2] that seeks to identify tests that reveal future failures in systems before they have occurred. A transformation that makes it easier to find anticipatory tests would help the tester to find such cases and may alleviate some of the challenges of anticipatory testing.

It will also be interesting to explore Anti-Testability Transformations; transformations that make the task of finding anticipatory tests *harder*. Using a more conventional meaning-preserving transformation approach (in which the transformed program is considered to be a meaning–preserving, and improved, version of the original), such anticipatory-test-denying transformations could yield a form of automated system security hardening.

6 Conclusion

This paper's primary role is to argue for a formal semantic underpinning for Testability Transformation. It outlines how such a semantic framework could combine practical industrial impact with theoretical elegance and intellectual challenge. The hope is that the research community will take up the challenge of defining these much-needed formal semantics.

The definition of the sets of semantics preserved by different test adequacy criteria will undoubtedly also shed much needed light on test adequacy more generally, and thereby provide a mathematical basis for much of the current *ad hoc* practices of software testing.

Acknowledgements. Many thanks to Patrick Cousot, Paul Marinescu, Peter O'Hearn, Tony Hoare, Mike Papadakis, Shin Yoo, and Jie Zhang for comments on earlier drafts. Thanks also to the Facebook Developer Infrastructure leadership for their support and to the European Research Council for part-funding my scientific work through the ERC Advanced Fellowship scheme.

References

1. Apt, K.R.: Ten years of Hoare's logic: a survey - part I. ACM Trans. Prog. Lang. Syst. **3**(4), 431–483 (1981)
2. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: Proceedings of the First European Systems Conference (EuroSys 2006), pp. 73–85. Leuven, Belgium, April 2006
3. Bardin, S., Delahaye, M., David, R., Kosmatov, N., Papadakis, M., Traon, Y.L., Marion, J.Y.: Sound and quasi-complete detection of infeasible test requirements. In: International Conference on Software Testing, Verification and Validation (ICST 2015), pp. 1–10. IEEE Computer Society (2015)

[2] https://www.pre-crime.eu.

4. Baresel, A., Binkley, D., Harman, M., Korel, B.: Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. In: International Symposium on Software Testing and Analysis (ISSTA 2004), pp. 108–118. Omni Parker House Hotel, Boston, Massachusetts, July 2004. Appears in Software Engineering Notes **29**(4)
5. Baresel, A., Sthamer, H.: Evolutionary testing of flag conditions. In: Cantú-Paz, E., Foster, J.A., Deb, K., Davis, L.D., Roy, R., O'Reilly, U.-M., Beyer, H.-G., Standish, R., Kendall, G., Wilson, S., Harman, M., Wegener, J., Dasgupta, D., Potter, M.A., Schultz, A.C., Dowsland, K.A., Jonoska, N., Miller, J. (eds.) GECCO 2003, Part II. LNCS, vol. 2724, pp. 2442–2454. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-45110-2_148
6. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The Oracle problem in software testing: a survey. IEEE Trans. Softw. Eng. **41**(5), 507–525 (2015)
7. Barraclough, R., Binkley, D., Danicic, S., Harman, M., Hierons, R., Kiss, A., Laurence, M.: A trajectory-based strict semantics for program slicing. Theor. Comput. Sci. **411**(11–13), 1372–1386 (2010)
8. Beizer, B.: Software Testing Techniques. Van Nostrand Reinhold, New York (1990)
9. Bertolino, A.: Software testing research: achievements, challenges, dreams. In: Briand, L., Wolf, A. (eds.) Future of Software Engineering 2007. IEEE Computer Society Press, Los Alamitos (2007)
10. British Standards Institute: BS 7925–2 software component testing (1998)
11. Cadar, C.: Targeted program transformations for symbolic execution. In: 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 906–909 (2015)
12. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
13. Cartwright, R., Felleisen, M.: The semantics of program dependence. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 13–27 (1989)
14. Chekam, T.T., Papadakis, M., Traon, Y.L., Harman, M.: An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, 20–28 May 2017, pp. 597–608 (2017)
15. Chen, T.Y., Feng, J., Tse, T.H.: Metamorphic testing of programs on partial differential equations: a case study. In: 26th Annual International Computer Software and Applications Conference (COMPSAC 2002), pp. 327–333. IEEE Computer Society (2002)
16. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Logic Comput. **2**(4), 511–547 (1992)
17. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation. In: The 29th ACM Symposium on Principles of Programming Languages (POPL 2002), pp. 178–190, Portland, Oregon, 16–18 January 2002
18. Darlington, J., Burstall, R.M.: A transformation system for developing recursive programs. J. Assoc. Comput. Mach. **24**(1), 44–67 (1977)
19. Goguen, J.A., Malcolm, G.: Algebraic Semantics of Imperative Programs. MIT Press, Cambridge (1996)
20. Gong, D., Yao, X.: Testability transformation based on equivalence of target statements. Neural Comput. Appl. **21**(8), 1871–1882 (2012)
21. Guttag, J.: Abstract data types and the development of data structures. Commun. ACM **20**(6), 396–404 (1977)

22. Harman, M.: Open problems in testability transformation (keynote paper). In: 1st International Workshop on Search Based Testing (SBST 2008), Lillehammer, Norway (2008)
23. Harman, M., Baresel, A., Binkley, D., Hierons, R., Hu, L., Korel, B., McMinn, P., Roper, M.: Testability transformation – program transformation to improve testability. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing. LNCS, vol. 4949, pp. 320–344. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78917-8_11
24. Harman, M., Danicic, S.: Using program slicing to simplify testing. Softw. Test. Verification Reliab. **5**(3), 143–162 (1995)
25. Harman, M., Hu, L., Hierons, R.M., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Trans. Softw. Eng. **30**(1), 3–16 (2004)
26. Harman, M., Jia, Y., Zhang, Y.: Achievements, open problems and challenges for search based software testing (keynote paper). In: 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2015), Graz, Austria, April 2015
27. Harman, M., Yao, X., Jia, Y.: A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: 36th International Conference on Software Engineering (ICSE 2014), pp. 919–930, Hyderabad, India, June 2014
28. Hierons, R., Harman, M., Fox, C.: Branch-coverage testability transformation for unstructured programs. Comput. J. **48**(4), 421–436 (2005)
29. Hoare, C.A.R.: An axiomatic basis of computer programming. Commun. ACM **12**, 576–580 (1969)
30. Jia, Y., Harman, M.: Higher order mutation testing. J. Inf. Softw. Technol. **51**(10), 1379–1393 (2009)
31. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
32. Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: International Symposium on Foundations of Software Engineering (FSE), pp. 654–665 (2014)
33. Kalaji, A., Hierons, R.M., Swift, S.: A testability transformation approach for state-based programs. In: 1st International Symposium on Search Based Software Engineering (SSBSE 2009), pp. 85–88. IEEE, Windsor, May 2009
34. Korel, B., Harman, M., Chung, S., Apirukvorapinit, P., Gupta, R.: Data dependence based testability transformation in automated test generation. In: 16th International Symposium on Software Reliability Engineering (ISSRE 2005), pp. 245–254, Chicago, Illinios, USA, November 2005
35. Kurtz, B., Ammann, P., Delamaro, M.E., Offutt, J., Deng, L.: Mutant subsumption graphs. In: 10th Mutation Testing Workshop (Mutation 2014), Cleveland Ohio, USA, March 2014, to appear
36. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: GenProg: a generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1), 54–72 (2012)
37. Li, Y., Fraser, G.: Bytecode testability transformation. In: Cohen, M.B., Ó Cinnéide, M. (eds.) SSBSE 2011. LNCS, vol. 6956, pp. 237–251. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23716-4_21
38. Madeyski, L., Orzeszyna, W., Torkar, R., Jozala, M.: Overcoming the equivalent mutant problem: a systematic literature review and a comparative experiment of second order mutation. IEEE Trans. Softw. Eng. **40**(1), 23–42 (2014)
39. McMinn, P.: Search-based failure discovery using testability transformations to generate pseudo-oracles. In: Rothlauf, F. (ed.) Genetic and Evolutionary Computation Conference (GECCO 2009), pp. 1689–1696. ACM, Montreal (2009)

40. McMinn, P., Binkley, D., Harman, M.: Empirical evaluation of a nesting testability transformation for evolutionary testing. ACM Trans. Softw. Eng. Methodol. **18**(3) (2009). Article no. 11
41. Papadakis, M., Jia, Y., Harman, M., Traon, Y.L.: Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: 37th International Conference on Software Engineering (ICSE 2015), pp. 936–946, Florence, Italy (2015)
42. Parsons-Selke, R.: A graph semantics for program dependence graphs. In: Sixteenth ACM Symposium on Principles of Programming Languages (POPL), Austin, TX, 11–13 January 1989, pp. 12–24 (1989)
43. Partsch, H.A.: The Specification and Transformation of Programs: A Formal Approach to Software Development. Springer, Heidelberg (1990)
44. Plotkin, G.D.: The origins of structural operational semantics. J. Logic Algebraic Prog. **60**, 3–15 (2004)
45. Radio Technical Commission for Aeronautics: RTCA DO178-B Software considerations in airborne systems and equipment certification (1992)
46. Reps, T., Yang, W.: The semantics of program slicing. Technical report 777, University of Wisconsin (1988)
47. Schulte, E., Fry, Z.P., Fast, E., Weimer, W., Forrest, S.: Software mutational robustness. Genet. Program. Evolvable Mach. **15**(3), 281–312 (2014)
48. Woodward, M.R., Halewood, K.: From weak to strong, dead or alive? An analysis of some mutation testing issues. In: 2nd Workshop on Software Testing, Verification, and Analysis. Banff, Canada, July 1988
49. Yu, Y.T., Lau, M.F.: A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. J. Syst. Softw. **79**(5), 577–590 (2006)
50. Zhang, J., Hao, D., Zhang, L., Zhang, L.: To detect abnormal program behaviours via mutation deduction. In: Mutation Testing Workshop, Mutation 2018, to appear

Specification

From Software Specifications to Constraint Programming

Stefan Hallerstede[1(✉)], Miran Hasanagić[1], Sebastian Krings[2], Peter Gorm Larsen[1], and Michael Leuschel[2]

[1] Department of Engineering, Aarhus University, Aarhus, Denmark
sha@eng.au.dk

[2] University of Düsseldorf, Düsseldorf, Germany

Abstract. Non-deterministic specifications play a central role in the use of formal methods for software development. Such specifications can be more readable, but hard to execute efficiently due to the usually large search space. Constraint programming offers advanced algorithms and heuristics for solving certain non-deterministic models. Unfortunately, this requires writing models in a form suitable for efficient solving where the readability typically required from a specification is lost. Tools like ProB attempt to bridge this gap by translating high-level first-order predicate logic specifications into formal models suitable for constraint solving. In this paper we study potential improvements to this methodology by (1) using refinement to transform specifications into models suitable for efficient solving, (2) translating first-order predicates directly into the OscaR framework and (3) using different kinds of solvers as a back end. Formal verification by proof ensures the correctness of the solution of the model with respect to the specification.

1 Introduction

State-based modelling methods like B and VDM support writing abstract specifications and implement them by refinement. Refinement is carried out semi-automatically, leading to deterministic implementations which are amenable to automatic code generation. A long term ambition of software tools like ProB [17] is to allow users to write very high-level specifications, which are easy to read and write, amenable to formal proof and can yet still be executed efficiently. In other words, specifications are viewed as non-deterministic programs, where constraint solving is used to compute solutions to their formal specifications [18]. Still, the efficient solution of hard constraint satisfaction problems often requires an encoding of formal descriptions of the problems that is difficult to comprehend. Such concerns turn specifications into non-deterministic implementations and open up the possibility for programming errors just as in the case of deterministic implementations. As a consequence, we benefit less from the declarative paradigm than we would expect. In this paper we propose to use formal refinement as a method to relate formal models *SPEC* that describe problems at a high level of abstraction to formal models *AIMP* at a lower level of abstraction targeting constraint

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 21–36, 2018.
https://doi.org/10.1007/978-3-319-92970-5_2

solvers. In the same way as for the deterministic case, refinement achieves correctness with respect to an easier to understand abstract specification. Figure 1 illustrates the approach. The low-level models can be translated into different frameworks like Constraint Programming (CP), Constraint Logic Programming over Finite Domains (CLP(FD)) or SAT for efficient execution according to their "flavour". The different models may still share modelling concepts but use specific concepts with efficient representations in the target frameworks. In order to ease formal refinement proofs, typically several refinement steps *REFN* are used before reaching an implementation (*AIMP* here).

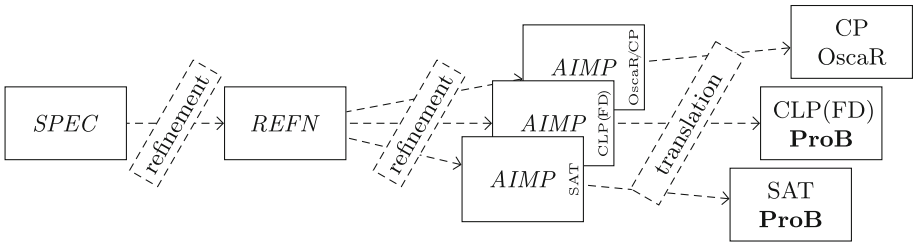


Fig. 1. Refinement and translation of abstract models

ProB [17] is a tool that can execute high-level models described in first-order predicate logic with set theory by translating them into different frameworks, in particular, CLP(FD) and SAT. We use the OscaR library [22] as an additional target bypassing ProB to experiment with different translations into a CP framework. The objective is to extend ProB eventually also with a CP translation. As a result of this approach ProB will be capable of providing an efficient target for the translation of a given model. Furthermore, we obtain a methodology to compare the performance of different frameworks as we can prove that they encode the same model while still exploiting their respective strengths. Finally, beside performance we are also interested in refutation completeness, in particular. Some search heuristics sacrifice refutation completeness in favour of performance. In some practical situations this is acceptable when the alternative is for the search to time out. Of course, for the user of a specification method it is essential to understand the significance of the answer obtained. In this paper we deal with the two latter points. We briefly discuss the translations provided by ProB and the one we use for OscaR. We compare the performance of the different translated models. Furthermore, since ProB can translate models at different levels of abstraction, we can compare performance gains achieved by refining models to lower levels of abstraction.

The following example illustrates the abstraction levels we have to bridge from a high-level problem description to an efficient CP encoding. We use the well-known n-queens problem [3] that is commonly used as a benchmark for constraint solvers and allows evaluating the scalability of our approach. Details are discussed in Sect. 4.

Example. Consider the following specification of the n-queens problem: n queens need to be placed on a n-times-n board such that no queen can attack another.

Let n be a (non-negative) integer constant. Let *NQABS* be the following predicate, a specification of the n-queens problem in first-order predicate logic:

$$\overbrace{b \subseteq 1\,..\,n \times 1\,..\,n}^{\textit{shape of the board}} \wedge \overbrace{\mathrm{card}\, b = n}^{\textit{n queens}} \wedge$$
$$\forall p, q \cdot p \mapsto q \in b \Rightarrow$$
$$\forall i \cdot i \neq 0 \Rightarrow \underbrace{p + i \mapsto q \notin b}_{\textit{horizontal}} \wedge \underbrace{p \mapsto q + i \notin b}_{\textit{vertical}} \wedge \underbrace{p + i \mapsto q + i \notin b}_{\text{“}\backslash\text{”-diagonal}} \wedge \underbrace{p + i \mapsto q - i \notin b}_{\text{“}/\text{”-diagonal}}$$

It captures the main characteristics of the problem: the shape of the board, the number of queens and a constraint specifying that no queen must be placed on the board such that it can be attacked by some queen $p \mapsto q \in b$ on the board horizontally, vertically or diagonally.[1]

The following specification *NQCON* is considered an abstract description when the problem is to be solved by means of constraint programming:

$$b \in \text{array } 1\,..\,n \text{ to } 1\,..\,n \wedge \text{allDifferent}\,(b) \wedge$$
$$\text{allDifferent}\,(\lambda x \cdot x \in 1\,..\,n \mid b(x) + x) \wedge \text{allDifferent}\,(\lambda x \cdot x \in 1\,..\,n \mid b(x) - x)$$

where $b \in \text{array } A \text{ to } B$ models an array and is defined to be a total function from A to B (formally, $\mathrm{dom}\, b = A \wedge \mathrm{ran}\, b \subseteq B \wedge b^{-1}\,;\, b \subseteq \mathrm{id}$) and allDifferent (b) for an array b as $b; b^{-1} \subseteq \mathrm{id}$, that is, b is injective. CP frameworks usually have optimised search heuristics for well-known constraints. See the study of efficient solvers for the `allDifferent` constraint in [13], for instance. After all, it requires some reasoning to see that any solution of the n-queens problem described by *NQCON* is a solution to the specification *NQABS*. Refinement bridges the gap between the two abstraction levels. It is tempting to ask for a stronger relationship than refinement by requiring that no solutions are lost in the implementation. However, this can be considered a design decision as one could, e.g., wish to remove certain “unwanted” solutions.

If the approach described in this article is followed, then specification and refinement give rise to a method of multi-paradigm programming where imperative, functional and logic program development styles are mixed. Program refinement integrates imperative and logic programming, whereas the theories used in the proofs contribute functional programming. All proofs presented in this article have been carried out with the Isabelle proof assistant [20] in Isabelle/HOL, that supports this view directly.

Related Work. In [10] the authors argue that specifications should be non-executable, being more expressive as well as providing a higher level of abstraction. Moreover, it is stated that executable specifications, being close to a programming language style, may introduce implementation bias together with over-specification. In general the distinction between these two specification styles is that a non-executable specification describes *what* to achieve, while the other captures *how* to achieve it. Hence, the two styles require a trade-off involving expressiveness against executability. A counterargument to [10] is discussed

[1] The term $p \mapsto q$ denotes the pair with first component p and second component q.

in [6] showing how abstract specifications can be written to be executable often even correspond closely to logic programs. The current literature addressing this gap focuses on translations from a formal model towards a constraint solver. For example [5,15] presents translations from Z to Prolog and VDM to ABC, respectively. However, such approaches compared to the methodology presented, focuses solely on making specifications executable, and do not consider combining refinement together with limiting the search space. In all of this work executable and non-executable specifications are opposed to each other. In our work specifications that are not readily executable are refined to executable ones. These may be non-deterministic or deterministic.

Overview of this Article. The remainder of this paper is organised as follows. Section 2 describes our approach of specification, refinement and constraint solving. It provides an example of an (admittedly simple) data refinement. This work focuses on methodological issues of combining different formal methods techniques and data refinement is one of the aspects we discuss. Section 3 discusses the translation of specifications into constraint programming languages. It is important to be aware of this because these languages directly affect the specifications that can be executed (just as is the case for other programming languages). In Sect. 4 we discuss the n-queens problem in greater depth. The n-queens problem is representative of combinatorial problems, in general. Whereas the example from Sect. 2 makes use of data refinement, the n-queens problem changes the structure of the specification during the refinement. Verified standard constraint programming models are derived by refinement. All the models are executed in ProB to witness the performance gains (or lack thereof). The final implementation is also executed in OscaR. The various models and proofs are available at https://github.com/miranha/SpecCP.

2 Refinement and Constraint Programming

Software specifications are described conveniently using predicate logic together with set theory, abstracting from details of data representation and program execution. They are abstract models of software. The discussion of refinement in this section follows conceptually the refinement notion of Event-B [1]. However, we avoid the discussion of Event-B proof obligations and focus on the joint use of refinement and constraint solving. We use first-order logic specifications with set theory and integer arithmetic. Isabelle/HOL, which is used to prove the refinement steps, is also used in other contexts to verify properties of Event-B and VDM specifications [4,25]. The examples that we use in this article, however, we have translated from and to Isabelle by hand into the dialect of the B-notation used by ProB, which can be achieved straightforwardly.

Refinement. In some situations abstract models *SPEC* can be implemented with reasonable effort as deterministic programs *PROG*. Specification *SPEC* is not as efficiently executable as the deterministic implementation *PROG*. Refinement permits us to improve *SPEC* gradually, introducing more implementation details step by step until reaching *PROG*. There are various notions of

refinement. Implication is one of them [11]: *REFN* refines *SPEC* if and only if $REFN \Rightarrow SPEC$. This notion can be extended to permit changes in the data representation, say, replacing variable v by variable w. The relationship is expressed by a predicate $SIMR(w, v)$. This is referred to as data refinement [24]. If $SIMR(w, v)$ is functional of the shape $v = SIMF(w)$, data refinement of $SPEC(v, v')$ by $REFN(w, w')$ can be described as follows:

$$w = SIMG(v) \wedge REFN(w, w') \wedge v' = SIMF(w') \Rightarrow SPEC(v, v')$$

where $w = SIMG(v) \Rightarrow v = SIMF(w)$. Now, *SIMR* is itself a specification and can be implemented (or it may be an implementation already). If *REFN*, *SIMF* and *SIMG* are executable, then we can compute *SPEC* in terms of them. In order to be able to understand the solution produced by the implementation we can use *SIMF* and *SIMG* to translate between the data representations. The refinement relation is considered part of the implementation and implemented itself. So, refinement is simply implication. For instance, we could sort an array a in terms of another representation b,

$$b = SIMG(a) \wedge REFN(b, b') \wedge a' = SIMF(b') \Rightarrow SPEC(a, a').$$

More generally, we refine a specification *SPEC* by a sequential program $TOCON;REFN;TOABS$, where *TOCON* translates into a suitable data representation and *TOABS* translates back. Figure 2 shows how *SPEC* is refined. In the examples treated in this article we only encounter functions of the sort *SIMF* and *SIMG* as described above.

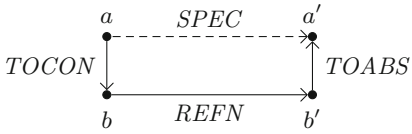


Fig. 2. Specification *SPEC* refined by specification $TOCON;REFN;TOABS$

This notion of refinement corresponds to guard refinement in Event-B where the concrete guard of an event (resp. a state transition) must imply the abstract guard [1]. It also corresponds to postcondition strengthening in VDM [14] or the refinement calculus [2,19]. In all of these cases a predicate is used to specify a successor state. In this article we focus on this predicate and use implication to express refinement. It can be applied easily in various model-based formal methods.

In some cases providing a deterministic implementation may be very difficult and constraint programming can be used. Instead of using refinement to provide algorithmic structure, it can be used to cast a predicate into a shape that can be solved efficiently by a constraint solver.

Example. We illustrate this by means of the puzzle *Who killed Agatha?* [23].

"Someone in Dreadsbury Mansion killed Aunt Agatha. Agatha, the butler, and Charles live in Dreadsbury Mansion, and are the only ones to live there.

A killer always hates, and is no richer than his victim. Charles hates no one that Agatha hates. Agatha hates everybody except the butler. The butler hates everyone not richer than Aunt Agatha. The butler hates everyone whom Agatha hates. No one hates everyone. Who killed Agatha?"

Assume we have three distinct constants *Agatha*, *butler* and *Charles*. We wish to determine the killer k in the following specification $WKA(r, h, k)$

$$\begin{array}{l} \mathit{Agatha} \in (h \setminus r)[\{k\}] \land \mathit{irreflexive}(r) \land \mathit{transitive}(r) \land \mathit{antisymmetric}(r) \land \\ h[\{\mathit{Agatha}\}] \cap h[\{\mathit{Charles}\}] = \varnothing \land h[\{\mathit{Agatha}\}] = P \setminus \{\mathit{butler}\} \land \\ (\forall x \cdot x \mapsto \mathit{Agatha} \notin r \Rightarrow \mathit{butler} \mapsto x \in h) \land h[\{\mathit{Agatha}\}] \subseteq h[\{\mathit{butler}\}] \land \\ (\forall x \cdot h[\{x\}] \neq P) \end{array}$$

where $P = \{\mathit{Agatha}, \mathit{butler}, \mathit{Charles}\}$, r models the relationship *richer* and h models the relationship *hates*. This specification captures the informal description above. The use of symbolic constants improves readability. However, searching for a solution for k this may not necessarily be the most efficient representation. Switching to integer constants $I = 0 \,..\, 2$ instead, we could use them in arithmetic expressions as in $\sum_{y \in h[\{x\}]} y$. We can map the values in I to values in P by means of the function

$$\mathit{abs}(i) = \text{if } i = 0 \text{ then } \mathit{Agatha} \text{ else (if } i = 1 \text{ then } \mathit{butler} \text{ else } \mathit{Charles})$$

and from P to I by means of the function

$$\mathit{con}(p) = \text{if } p = \mathit{Agatha} \text{ then } 0 \text{ else (if } p = \mathit{butler} \text{ then } 1 \text{ else } 2).$$

Note that, $i = \mathit{con}(p) \Rightarrow p = \mathit{abs}(i)$. We can prove that $WKB(s, j, \ell)$ given by

$$\begin{array}{l} 0 \in (j \setminus s)[\{\ell\}] \land \mathit{irreflexive}(s) \land \mathit{transitive}(s) \land \mathit{antisymmetric}(s) \land \\ j \subseteq I \times I \land s \subseteq I \times I \land j[\{0\}] \cap j[\{2\}] = \varnothing \land j[\{0\}] = I \setminus \{1\} \land \\ (\forall x \cdot x \in I \land x \mapsto 0 \notin s \Rightarrow 1 \mapsto x \in j) \land j[\{0\}] \subseteq j[\{1\}] \land \\ (\forall x \cdot x \in I \Rightarrow j[\{x\}] \neq I) \end{array}$$

refines $WKA(r, h, k)$. Formally,

$$\begin{array}{l} s = \{\mathit{con}(x) \mapsto \mathit{con}(y) \mid x \mapsto y \in r\} \land j = \{\mathit{con}(x) \mapsto \mathit{con}(y) \mid x \mapsto y \in h\} \land \\ WKB(s, j, \ell) \land k = \mathit{abs}(\ell) \Rightarrow WKA(r, h, k). \end{array}$$

Note, that the equations in the first line describe functions. Finally, we arrive at a shape of the specification $WKC(s, j, \ell)$ that permits efficient execution by a constraint solver,

$$\begin{array}{l} \ell \in I \land 0 \mapsto \ell \in j^{-1} \land 0 \mapsto \ell \notin s^{-1} \land \\ \mathit{irreflexive}(s) \land \mathit{transitive}(s) \land \mathit{antisymmetric}(s) \land \\ j \subseteq I \times I \land s \subseteq I \times I \land (\forall x \cdot x \in I \land 0 \mapsto x \in j \Rightarrow 2 \mapsto x \notin j) \land \\ 0 \mapsto 0 \in j \land 0 \mapsto 1 \notin j \land 0 \mapsto 2 \in j \land (\forall x \cdot x \in I \land x \mapsto 0 \notin s \Rightarrow 1 \mapsto x \in j) \land \\ (\forall x \cdot x \in I \land 0 \mapsto x \in j \Rightarrow 1 \mapsto x \in j) \land (\forall x \cdot x \in I \Rightarrow \sum_{y \in j[\{x\}]} 1 \leq 2) \end{array}$$

To translate this formula into a dedicated constraint programming model, we would next replace some formulas to match with the library of the constraint programming language. For example, we would replace $\sum_{y \in j[\{x\}]} 1$ by $\text{sum}(j[\{x\}])$ in the last row. Furthermore, we would replace the relations j and s by corresponding two-dimensional boolean arrays. However, this representation is a trivial rewriting of the formula above, replacing terms of the shape $(x, y) \in z$ by $a(x, y) = \text{TRUE}$, and we do not show it. Figure 3 shows how the model for the puzzle could be translated to OscaR. The translation has been produced by hand following the rules described in Sect. 3. It consists of a section of declarations, the actual model and a call to the solver. The most interesting part for this article is the development of models that can be analysed efficiently. Note, the use of `transpose` in the OscaR model. This is an implementation concern that is reflected by the use of the relational inverse in the abstract formula: the first index of `j` and `s` must not be a `CPIntVar`.

```
val I = 0 to 2                                          ┐
val l = CPIntVar(I)                                     │
val j = Array.fill(3,3)(CPBoolVar())                    │ Declarations
val s = Array.fill(3,3)(CPBoolVar())                    ┘
{val t = j.transpose; add(t(0)(l).isEq(1))}             ┐
{val t = s.transpose; add(t(0)(l).isEq(0))}             │
irrefl(I, s); trans(I, s); antisym(I, s)                │
I.foreach(i=>add((j(0)(i)) ==> (!j(2)(i))))             │ Model
add(j(0)(2)); add(!j(0)(1)); add(j(0)(0))               │
I.foreach(i=>add(!s(i)(0) ==> j(1)(i)))                 │
I.foreach(i=>add(j(0)(i) ==> j(1)(i)))                  │
I.foreach(i=>add(sum(j(i)) <= 2))                       ┘
search{                                                 ┐
  binaryFirstFail(Seq(l))                               │ Solver
}                                                       ┘
```

Fig. 3. OscaR model with solver for the puzzle "Who killed Agatha?"

The constraint solver of ProB can solve the models at all abstraction levels that we have presented in this section. A set like $P = \{Agatha, butler, Charles\}$ gets translated by ProB internally into the set $\{1, 2, 3\}$; as such the first data refinement *WKB* is performed internally by ProB and not required by the user. The translation of the constraint $\forall x \cdot h[\{x\}] \neq P$ into a sum constraint in *WKC* is also not necessary: ProB has built-in support for set equality and inequality and can handle the set inequality $h[\{x\}] \neq P$ relatively efficiently.

3 Translation and Constraint Solving

This section discusses the translation approaches introduced in Sect. 1. Section 3.1 discusses the use of specifications as constraint solving languages directly with ProB, outlining some important concepts of the applied translations. Section 3.2 outlines the translation of predicate logic statements from a

specification into a language for constraint solving in OscaR. Whereas the translation to OscaR requires the user to adopt certain data representations, ProB can also solve the abstract specifications. In fact, internally ProB attempts to find efficient representations to increase the speed of the search. However, no generally efficient method exists [12]. A longer term goal of our work combining refinement with translations is to permit writing specifications that steer the internal representations to gain more control over the efficiency while keeping the abstraction level high. Independently of this, the approach used by ProB for finding the solution to a non-deterministic specification influences the specification style for specific problems. In other words, implementation concerns always shine through.

3.1 Translation in ProB

A key concept of ProB's default constraint solver [8] is reification, i.e., representing the truth value of a constraint C by a boolean decision variable $R_C \in 0..1$ so that $R_C = 1 \Leftrightarrow C$. Reification is important for ProB to avoid choice points, e.g., for $P \vee Q$, ProB will set up the constraint similar to $R_P \in 0..1 \wedge (R_P = 1 \Leftrightarrow P) \wedge R_Q \in 0..1 \wedge (R_Q = 1 \Leftrightarrow Q) \wedge (R_P = 1 \vee R_Q = 1)$. The constraint $(R_P = 1 \vee R_Q = 1)$ is handled by ProB's boolean constraint solver, while P and Q can be handled by different solvers. Indeed, arithmetic predicates and operators like $x + y \geq 0$ are mapped to the finite domain solver CLP(FD). Equality, inequality, set membership and subset constraints are handled by a dedicated solver in ProB itself.

In the worst case, universal quantifiers $\forall x.Q \Rightarrow R$ get expanded when the domain $\{x|Q\}$ is known. However, there is special support for $\forall x.x \in S \Rightarrow P(x)$: here P will be checked for every element added to S, even when S is not yet fully known. Certain universal quantifiers can be expanded into a conjunction: e.g., $\forall x.x \in 1..3 \Rightarrow P(x)$ gets automatically translated into $P(1) \wedge P(2) \wedge P(3)$; this enables reification of the entire quantified predicate. The treatment of existential quantifiers is similar; in the worst case they are evaluated when all but the quantified variables are known. However, some quantifiers can be expanded into disjunctions: e.g., $\exists x.x \in 1..3 \wedge P(x)$ gets translated into $P(1) \vee P(2) \vee P(3)$.

As mentioned earlier, ProB provides alternate constraint solving backends: a translation to SAT via Kodkod and a translation to SMTLib using Z3. The SAT encoding via Kodkod performs well for constraints over relations and operators such as relational composition and transitive closure. It, however, requires all base types to be finite. For the "Who killed Agatha?" of Sect. 2 the SAT backend is slightly slower (60 ms vs 10 ms). For the n-queens problem in Sect. 4, it is considerably slower than ProB's default solver (e.g., over 5 s compared to 10 ms for $n = 16$). The Z3 backend is even slower (e.g., over 6 s for $n = 8$). Hence, in the rest of the paper we have concentrated on the default solver of ProB.

3.2 Translation to OscaR

OscaR (using many ideas of Comet [12]) supports the execution of non-deterministic specifications, as well as permitting us to experiment easily with different model representations and search heuristics. Furthermore, OscaR is a library for Scala [21], so the used syntax is Scala syntax with the extensions made by OscaR. A specification $\forall z \cdot p_1(x,z) \wedge \ldots p_n(x,z)$ specifies values of variables x to be computed, constrained by relationships among each other and variables z. Universal quantifiers and conjunctions are preferred over existential quantifiers and disjunctions as the latter may lead to backtracking. However, some uses do lead to loss of efficiency and sometime even lead to performance improvements. For instance, we use auxiliary variables to add additional constraints (sometimes redundant but with an effect on performance). So a specification has the form $\exists y \cdot \forall z \cdot p_1(x,y,z) \wedge \ldots p_n(x,y,z)$ where y corresponds to an auxiliary variable of the corresponding constraint program. Generally, OscaR tracks domains of variables efficiently, while the search algorithms restrict the domains further in order to minimise the remaining non-deterministic choices. By means of backtracking OscaR is able to find multiple solutions. In comparison to OscaR, ProB is able to find solutions for certain infinite problems and it will find all solutions for finitely posed problems. For OscaR the latter property depends on the chosen search heuristics. However, usually completeness is sacrificed for efficiency.

The OscaR CP solver adopts a modelling methodology using decision variables and constraints among them, similar to other CP solvers. Its general structure is based around three components as shown in Fig. 3: a declaration of the decision variables, a model and a search heuristic. The *declaration* introduces the decision variables and their domains. The *model* component captures the constraints for the decision variables. The *search heuristic* specifies a non-deterministic search heuristic for finding a solution for the model. Usually the declaration and model together are considered to be one component. We treat them separate as this is relevant for the translation. The domains associated with the different variables are highly relevant for the efficiency of the search heuristic. We only discuss the translation for the declaration and the model, and assume that a common search heuristic is to be applied to the model, such as *binary first fail*. The translation is indicated by the notation $S \overset{cp}{\rightsquigarrow} P$ describing the translation from specification element S to constraint programming construct P. We describe the translation by way of examples. They are easy to understand and generalise. For these translations the focus is on the extensions relevant for a specification as provided by OscaR.

The Declarations. The basic data types allowed for decision variables supported by OscaR are boolean and integer types. An integer type `CPIntVar` must be given a finite domain from which its values may be drawn. The boolean type `CPBoolVar` is a subtype of integer with the domain of 0 and 1. The translation of integer and boolean domains as well as declaration of decision variables can be translated as follows, respectively: $D = m \mathinner{..} n \overset{cp}{\rightsquigarrow}$ `val D = m to n`, $i \in D \overset{cp}{\rightsquigarrow}$ `val i = CPIntVar(D)` and $b \in \mathbb{B} \overset{cp}{\rightsquigarrow}$ `val b = CPBoolVar()`.

The Model. Once the decision variables have been declared together with their domain, constraints can be added to the constraint store. In OscaR constraints are added to the constraint store by means of the function `add(...)`. When adding constraints, two important aspects of the constraints to consider are conjunctions and disjunctions. Although both are of boolean type, they are treated differently by OscaR to improve the performance of the search for a solution. Atomic predicates are translated by $P \overset{cp}{\rightsquigarrow}$ `add(`$\mathtt{P}^{cp}$`)` where $\mathtt{P}^{cp}$ is the atomic predicate in OscaR syntax, for instance, the translation of $x \leq y$ is `x.isLeEq(y)`.

A conjunction describes a collection of constraints each of which needs to be true. They can be added separately to the constraint store. For example, $x = 5 \wedge y \leq x \overset{cp}{\rightsquigarrow}$ `add(x.isEq(5)); add(y.isLeEq(x))`. This is the common approach for treating conjunctions in OscaR. The general form of this is of the following shape $P \wedge Q \overset{cp}{\rightsquigarrow} P'; Q'$ where P and Q are translated to P' and Q'.

A disjunction describes a collection of constraints either of which needs to be true. It cannot be divided into separate constraints as is the case with conjunction. All separate constraints are implicitly conjoined. For this reason disjunctions are represented as boolean expressions within the constraint language. The operators available for this are logical or (`||`), logical and (`&&`) as well as implication (`==>`). This lifts the predicate into a constraint expression of the OscaR constraint language. This is applied to the disjunction translation, for example, $x = 5 \vee y \leq x \overset{cp}{\rightsquigarrow}$ `add(x.isEq(5) || y.isLeEq(x))`, while the general form is $P \vee Q \overset{cp}{\rightsquigarrow}$ `add(`$\mathtt{P}^{cp}$ `||` $\mathtt{Q}^{cp}$`)` where $\mathtt{P}^{cp}$ and $\mathtt{Q}^{cp}$ are translations of the conjuncts into OscaR syntax. Technically, $\mathtt{P}^{cp}$ and $\mathtt{Q}^{cp}$ require a second layer in the translation for constraint expressions but we do not develop this here. We make the strong assumption that the disjuncts are atomic. This introduces an implementation concern into the constraint modelling language. It is justified by the following reasoning: representing operators within constraints creates an additional layer for the search reducing its performance.[2] So the larger portion of a specification contains such constructs, the lower the performance will be. Hence, generally it is discouraged to represent large parts of a constraint program as constraint expressions because this negatively impact implementation performance. Nonetheless, the connectives `||`, `&&` and `==>` are very useful in OscaR (see Fig. 3) but should be used as little as possible. In particular, for conjunction it can usually be avoided.

We could add the rule $P \wedge Q \overset{cp}{\rightsquigarrow}$ `add(`$\mathtt{P}^{cp}$ `&&` $\mathtt{Q}^{cp}$`)` to the translation, e.g. to deal with cases where conjunctions appear within disjunctions. However, we are more interested to use refinement to arrive at efficient representations than at maximising the class of translatable specifications. In any case, ProB which is described above permits a large class of translations already. The objective of the translation to OscaR is to achieve high performance. Generality is secondary. Whenever possible, disjunctions should be avoided. For instance, instead of writing a specification $x = 1 \vee x = 2 \vee x = 3$ one can declare the domain of x correspondingly. If that is not possible, one could also introduce a deci-

[2] Private communication with Pierre Schaus.

sion variable y with domain $1 \,..\, 3$ and use the constraint $x = y$. Computations concerning constraints are very efficient! Of course, there will be cases where disjunctions are unavoidable. Negation is available because the domains of the involved decision variables are declared. As a result, negation may never lead to infinite domains rendering the model non-executable. The specification must contain a term $x \in D$ providing the (finite) domain for each decision variable x. And each universal quantification must provide a (finite) domain, that is, it must be of the shape $\forall x \cdot x \in D \Rightarrow P$. Universal quantification is discussed next.

A universal quantifier can be translated by adding individually each constraint described by way of the quantified variable with finite domain. For example, $\forall x \cdot x \in 4 \,..\, 7 \Rightarrow x > 3 \overset{cp}{\rightsquigarrow}$ `(4 to 7).foreach(x => add(x>3))`. The Scala operator `foreach` iterates through all elements of the range (here `(4 to 7)`). The general form of the translation has the shape $\forall x \cdot x \in m \,..\, n \Rightarrow P(x) \overset{cp}{\rightsquigarrow}$ (`mton`).`foreach`(`x` => `P'`(`x`)), where $P'(x)$ is the translation of $P(x)$. Universal quantification is essentially treated like an indexed conjunction.

Finally, existential quantifiers are discussed next. If existential terms are involved in a specification they must be lifted so that the specification has the shape $\exists y \cdot p_1(x, y) \wedge \ldots p_n(x, y)$ so that they can be treated as auxiliary variables. The translation does not support existential quantifiers occurring inside universal quantifiers. OscaR does not support this directly. In principle, the translation could deal with this in the way ProB does (see the discussion in Sect. 3.1). Whereas in the case of the universal quantifier the bound variable gets eliminated in the translation, in the case of existential quantifiers they stay. They are treated like the global variables but are not considered part of the result. As indicated in the introduction of this section, they should be considered auxiliary variables.

4 Solving the N-Queens Problem

In the example "Who killed Agatha?" of Sect. 2 we demonstrated the change of data representation. This permitted us to express a subset relationship as an arithmetic expression. The problem statement remained structurally unchanged: it is easy to see how the computed solution is related to the abstract specification.

In this section we change the specification structurally while keeping the data representation by way of the n-queens problem. The encodings of the n-queens problem are not new. Usually, one finds informal arguments that argue why a certain encoding is correct (e.g. [12]). However, the smarter the encodings get, the more likely errors occur in the corresponding models. This is no different from the situation in sequential programming. Finally, we evaluate the various models and compared there scalability.

Refinement of n-queens. The initial specification describes the problem in terms of the geometry of the chess board. Numbering the rows and columns from 1 to n for some n larger than 0, we can describe the queens on which fields a queen could attack arithmetically as indicated in Fig. 4. We can express that a

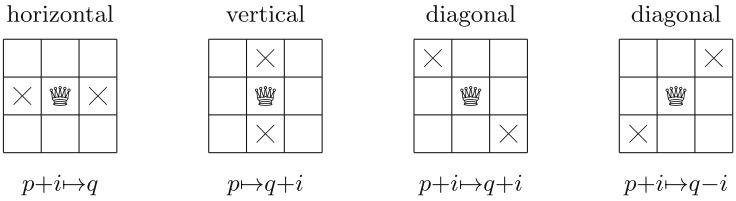


Fig. 4. Positions "×" on 3×3-board that can be attacked by a queen "♕" on position $p \mapsto q$ (where $i \in \{-1, 1\}$)

queen at position $p \mapsto q$ cannot attack another queen by requiring that no queen may be placed on a position it may reach, that is, $p \mapsto q \in b$ implies

$$\forall i \cdot i \neq 0 \Rightarrow p+i \mapsto q \notin b \wedge p \mapsto q+i \notin b \wedge p+i \mapsto q+i \notin b \wedge p+i \mapsto q-i \notin b,$$

that no other queen is on a position that can be attacked by it. The complete specification *NQABS* also specifying the board positions $1 \mathinner{..} n \times 1 \mathinner{..} n$ and the number of queens n to be placed on the board is easy to understand,

$$b \subseteq 1 \mathinner{..} n \times 1 \mathinner{..} n \wedge \operatorname{card} b = n \wedge \forall p, q \cdot p \mapsto q \in b \Rightarrow \\ \forall i \cdot i \neq 0 \Rightarrow p+i \mapsto q \notin b \wedge p \mapsto q+i \notin b \wedge p+i \mapsto q+i \notin b \wedge p+i \mapsto q-i \notin b$$

and relate to the informal statement of the n-queens problem. It relates to the board geometry, the number of queens and the positions a queen placed on the board may attack. Unfortunately, this specification is not efficient to execute. The first problem we identify is the size of the board to consider. It is n^2. Because a queen can attack any other queen that is placed in the same column we can exclude all configurations of queens on a board where more than one queen is on any column. Thus, there must be precisely one queen on each column. Hence, we can represent the board as an array in a first refinement *NQARR*

$$b \in \text{array } 1 \mathinner{..} n \text{ to } 1 \mathinner{..} n \wedge \\ \forall p, q \cdot p \mapsto q \in b \Rightarrow \forall i \cdot i \neq 0 \Rightarrow p+i \mapsto q \notin b \wedge p+i \mapsto q+i \notin b \wedge p+i \mapsto q-i \notin b$$

Of course, it is no longer necessary to verify that at most one queen placed in each column. The predicate $\forall p, q \cdot p \mapsto q \in b \Rightarrow \forall i \cdot i \neq 0 \Rightarrow p+i \mapsto q \notin b$ is a slightly complicated way of saying that b is injective. This is expressed by the formula allDifferent (b). When translating to constraint programming languages like OscaR, predicates like allDifferent are translated into efficient representations in the constraint store. Typically, a library of such predefined constraint predicates exists in constraint programming languages that permit performance improvements when used. The only predicate that could still be improved is $\forall p, i \cdot i \neq 0 \Rightarrow p+i \mapsto b(p)+i \notin b \wedge p+i \mapsto b(p)-i \notin b$ where we have used that b is a total function from $1 \mathinner{..} n$ to $1 \mathinner{..} n$ to rewrite the predicate.

We have

$$\begin{aligned}
& \forall p, i, q \cdot i \neq 0 \Rightarrow p + i \mapsto b(p) + i \neq q \mapsto b(q) \wedge p + i \mapsto b(p) - i \neq q \mapsto b(q) \\
\Leftarrow\ & \forall p, i, q \cdot i \neq 0 \Rightarrow i \mapsto i \neq q - p \mapsto b(q) - b(p) \wedge i \mapsto -i \neq q - p \mapsto b(q) - b(p) \\
\Leftarrow\ & \forall p, i, q \cdot i \neq 0 \Rightarrow i \mapsto i \neq \mathrm{abs}\,(q - p) \mapsto \mathrm{abs}\,(b(q) - b(p)) \\
\Leftarrow\ & \forall p, q \cdot p \neq q \Rightarrow \mathrm{abs}\,(q - p) \neq \mathrm{abs}\,(b(q) - b(p)) \qquad (1)
\end{aligned}$$

Hence, *NQARR* is refined by *NQMID* which we define by

$$\begin{aligned}
& b \in \text{array } 1 \,..\, n \text{ to } 1 \,..\, n \wedge \\
& \text{allDifferent (b)} \wedge \forall p, q \cdot p \neq q \Rightarrow \mathrm{abs}\,(q - p) \neq \mathrm{abs}\,(b(q) - b(p))
\end{aligned}$$

Specification *NQMID* is often used in examples for constraint programming. Furthermore, we have $(\forall p, q \cdot p \neq q \Rightarrow b(p) + p \neq b(q) + q \wedge b(p) - p \neq b(q) - q) \Rightarrow (1)$. Thus we can refine *NQMID* by *NQCON*, defined by

$$\begin{aligned}
& b \in \text{array } 1 \,..\, n \text{ to } 1 \,..\, n \wedge \text{allDifferent}\,(b) \wedge \\
& \text{allDifferent}\,(\lambda x \cdot x \in 1 \,..\, n \mid b(x) + x) \wedge \text{allDifferent}\,(\lambda x \cdot x \in 1 \,..\, n \mid b(x) - x)
\end{aligned}$$

The predicate allDifferent (a) is widely used in constraint programming. Informally, it is defined as: no value occurs more than once in the array a.

A formal refinement verifies that any solution computed for a refined specification is also a solution of the initial specification. In addition, the proofs provide an explanation for the correctness of the encodings.

Note that the way we have proved the refinements, we have not assured the existence of a solution in the refined specifications. We could refine any specification by *false*. One could prove the existence of a solution for each specification. However, for specific values of n this is what a constraint solvers does: it finds values for the decision variables that satisfy the specification.

Performance of the Specifications. We evaluate the performance of all four specifications *NQABS*, *NQARR*, *NQMID* and *NQCON*, where ProB can solve all while OscaR is only applied to *NQCON* using its built-in `allDifferent` constraint. Additionally for ProB, we add *NQPRM* which uses a built-in predicate for permutations and a random search heuristic and is otherwise identical to *NQCON*. The specifications are evaluated for various board sizes ranging from $n = 8$ to $n = 1000$, with a step size of 8. Benchmarks were run on AMD Opterons with 2 GHz and four physical cores; up to three benchmarks were run in parallel. Measurements show the time in seconds taken to find one solution for the corresponding specification of the board size n. Results are shown in Fig. 5, where missing combinations of boards size and solver are due to time-outs, i.e., the solver failed to produce an answer in 5 min.

All in all, benchmark results support our motivation to translate specifications to OscaR, as OscaR allows to explorate much larger models. First, note that the most abstract specification *NQABS* only scales to $n = 8$. Next, the more low-level *NQARR* and its following refinement *NQMID* display similar performance and scale to at most $n = 168$. The least abstract model *NQCON*

scales to $n = 600$, while the random permutation version *NQPRM* only scales slightly better. Finally, the OscaR version solves a board size of up to $n = 1000$.

Additionally, note that OscaR displays a somewhat erratic behaviour, e.g., it is slower or cannot find a solution for certain smaller board sizes, while being able to solve larger ones more efficiently. This might be due to both the internal implementation of OscaR and the `allDifferent` constraint, allowing to exploit symmetries for certain board sizes. For example the search of the state space might be vulnerable to wrong choices, such that it might has to explore a large sub-tree before getting on the right track again. With techniques such as conflict driven clause learning [27] and random restarts [7], the search can become much more resilient. In this paper OscaR and ProB are used as representatives of a CP solver, and the presented method can consider others as well. For this purpose, the n-queens problem served well as a simple, yet scalable, benchmark for evaluating the methodology presented. As we have argued in [16], more involved benchmarks allow for higher transferability of the benchmarks to real-world applications. In this respect, in [9,26] we discuss real-world applications, where constraint solving is used for solving and validating larger timetables and railway network configurations.

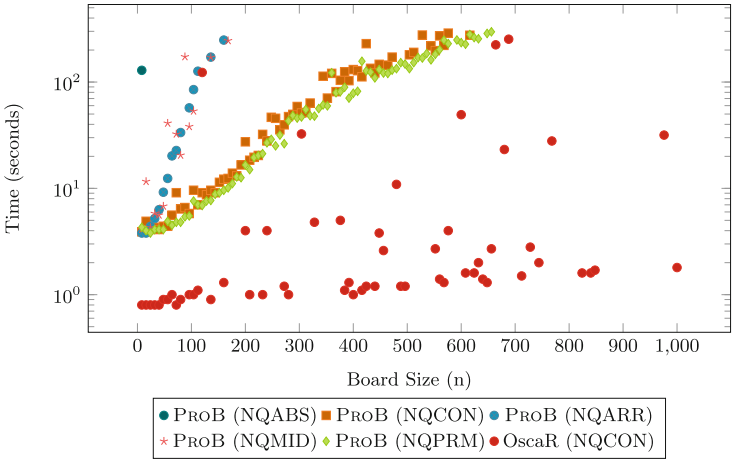


Fig. 5. Results of successful benchmarks, time-outs omitted

5 Conclusion

We have shown how refinement permits work with executable specifications without committing early to implementation-biased data representations. Formal specification should be considered abstract programming and associated reasoning techniques like refinement support for implemention. In principle, this software is multi-paradigm: it may contain imperative, logic and functional parts that appear seamless in abstract specifications. Compared to work like [19] where *programs* are executable or non-executable, we change the focus once more and

say that *specifications* are executable or non-executable to emphasise the abstraction also in the final implementation. We have described a translation to OscaR that we use for experimenting with translations and search heuristics. This complements the use of ProB where making consistent changes to the translation or search is more intricate. The preliminary evaluation to compare efficiency of abstraction levels points towards the direction in which this research will be continued: specifications are the better programs!

Acknowledgments. The work presented here is partially supported by the INTO-CPS project funded by the European Commission's Horizon 2020 programme under grant agreement number 664047.

References

1. Abrial, J.R.: Modeling in Event-B – System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer, New York (1998). https://doi.org/10.1007/978-1-4612-1674-2
3. Bruen, A., Dixon, R.: The n-queens problem. Discrete Math. **12**(4), 393–395 (1975)
4. Couto, L.D., Foster, S., Payne, R.J.: Towards verification of constituent systems through automated proof. CoRR abs/1404.7792 (2014)
5. Dick, A.J.J., Krause, P.J., Cozens, J.: Computer aided transformation of Z into prolog. In: Nicholls, J.E. (ed.) Z User Workshop. Workshops in Computing, pp. 71–85. Springer, London (1990). https://doi.org/10.1007/978-1-4471-3877-8_5
6. Fuchs, N.E.: Specifications are (preferably) executable. Softw. Eng. J. **7**(5), 323–334 (1992)
7. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting combinatorial search through randomization. In: Mostow, J., Rich, C. (eds.) AAAI, pp. 431–437. AAAI Press/MIT Press (1998)
8. Hallerstede, S., Leuschel, M.: Constraint-based deadlock checking of high-level specifications. TPLP **11**(4–5), 767–782 (2011)
9. Hansen, D., Schneider, D., Leuschel, M.: Using B and ProB for data validation projects. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 167–182. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_10
10. Hayes, I., Jones, C.B.: Specifications are not (necessarily) executable. Softw. Eng. J. **4**(6), 330–338 (1989)
11. Hehner, E.C.R.: A Practical Theory of Programming. Springer, New York (1993). https://doi.org/10.1007/978-1-4419-8596-5
12. Hentenryck, P.V., Michel, L.: Constraint-Based Local Search. MIT Press, Cambridge (2009)
13. van Hoeve, W.J.: The all different constraint: a survey. arXiv cs/0105015 (2001)
14. Jones, C.B.: Systematic Software Development Using VDM. Prentice Hall, Englewood Cliffs (1990)
15. Kans, A., Hayton, C.: Using ABC to prototype VDM specifications. SIGPLAN Not. **29**(1), 27–36 (1994)
16. Krings, S., Leuschel, M., Körner, P., Hallerstede, S., Hasanagić, M.: Three is a crowd: SAT, SMT and CLP on a chessboard. In: Calimeri, F., Hamlen, K., Leone, N. (eds.) PADL 2018. LNCS, vol. 10702, pp. 63–79. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73305-0_5

17. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. STTT **10**(2), 185–203 (2008)
18. Leuschel, M., Schneider, D.: Towards B as a high-level constraint modelling language – solving the jobs puzzle challenge. In: Ameur, Y.A., Schewe, K.D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 101–116. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_8
19. Morgan, C.C.: Programming From Specifications, 2nd edn. Prentice Hall, Englewood Cliffs (1994)
20. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
21. Odersky, M., al.: An Overview of the Scala Programming Language. Technical report. IC/2004/64, EPFL, Lausanne, Switzerland (2004)
22. OscaR Team: OscaR: Scala in OR (2012). bitbucket.org/oscarlib
23. Pelletier, F.J.: Seventy-five problems for testing automatic theorem provers. J. Autom. Reason. **2**, 191–216 (1986)
24. de Roever, W.P., Engelhardt, K.: Data Refinement: Model-oriented Proof Theories and their Comparison, vol. 46. Cambridge University Press, Cambridge (1998)
25. Schmalz, M.: Term rewriting in logics of partial functions. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 633–650. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24559-6_42
26. Schneider, D., Leuschel, M., Witt, T.: Model-based problem solving for university timetable validation and improvement. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 487–495. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_30
27. Silva, J.P.M., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999)

Automated Specification Extraction and Analysis with Specstractor

Christoph Schulze[1,2], Rance Cleaveland[1(✉)], and Mikael Lindvall[2]

[1] Department of Computer Science, University of Maryland, College Park, MD 20742, USA
cschulze@umd.edu, rance@cs.umd.edu

[2] Fraunhofer Center for Experimental Software Engineering, College Park, MD 20742, USA
mikli@fc-md.umd.edu

Abstract. This paper presents Specstractor, a tool chain for the extraction and analysis of system specifications in the form of collections of invariants. Such invariants convey valuable information about the behavior of a software system and are also useful in identifying missing or defective parts of existing specifications. Using data-mining techniques, Specstractor derives likely invariants from test data that it automatically generates from the system under analysis, using an iterative approach to refine the set of proposed invariants and eliminate false positives. The paper describes the Spectstractor technology and evaluates it on real-world artifacts from automotive-control and medical-device applications.

1 Introduction

Specstractor is an automated tool chain for the extraction and analysis of proposed system invariants from testing data. These invariants yield useful insight into the actual system behavior and can reveal issues in developer-maintained specifications. They can also serve as a starting point for formal system specifications when such specifications do not exist. Invariants have long been used in software specification and development, as they can convey important information about the relationships among state variables that must be preserved throughout behaviors of a software system and can do so at a higher level of abstraction than system code [1–4].

This paper presents the foundations, design and implementation of Specstractor. The tool is intended to work on so-called *Mealy systems*, which are formally defined in this paper, and which Simulink models may be seen as instances of. The tool uses data mining over test cases generated from Mealy systems to propose invariants. Since data-mining techniques rely on data that may only give a partial view of system behavior, some of the invariants Specstractor infers can be invalid. In order to combat these false positives the tool employs an iterative *test* → *infer* → *instrument* → *retest* paradigm. Specifically, an initial set of invariants is inferred from the first set of automatically generated test cases.

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 37–53, 2018.
https://doi.org/10.1007/978-3-319-92970-5_3

Additional testing cycles are used to identify false positives and to refine the proposed set of invariants until a final, stable set of invariants has been found. Specstractor also employs static analyses of the system to improve further the quality of the resulting invariants and the performance of the data-mining algorithms [5]. The tool is designed in a modular fashion that allows users to add support for new test generators and data-mining technologies.

This paper also describes an addition to the core functionality of Specstractor: the capability to infer invariants involving numerically-valued variables. Other work on invariant mining [1,5] required variables to come from enumerated types; they could not infer an invariant such as `speed` $> 25 \Rightarrow$ `cruiseControl` $=$ `off` without manual abstractions supplied by the user. To provide better support for numerical variables our Range Miner algorithm uses a combination of automated data binning, association-rule mining and merging and adjustment of the invariants to generate accurate value ranges within invariants. The paper concludes with an evaluation of Specstractor and Range Miner on Simulink models of automotive applications and medical devices.

2 Mathematical Preliminaries

This section introduces the class of system models, *Mealy systems*, and invariants, *transition specifications*, that the Specstractor framework works with.

2.1 Mealy Machines

Definition 1. *A Mealy machine is a tuple* $\langle Q, Q_0, I, O, \delta \rangle$ *where:*

1. Q *is a non-empty set of* states*;*
2. $Q_0 \subseteq Q$ *is the non-empty set of* start states*;*
3. I *is the set of* input symbols*;*
4. O *is the set of* output symbols*; and*
5. $\delta \in (Q \times I) \to (O \times Q)$ *is the* transition function*.*

Traditional Mealy machines also require that Q, I and O be finite and that Q_0 be a singleton set, but we do impose these restrictions in this paper. A Mealy machine represents a system whose states are given by Q. When the machine is in a given state, δ is used to compute the new state based on inputs given by the system's environment, with the computed outputs being delivered to the environment as a result. We use the following definitions in what follows.

Definition 2. *Let* $M = \langle Q, Q_0, I, O, \delta \rangle$ *be a Mealy machine.*

1. $\langle q, i, o, q' \rangle \in Q \times I \times O \times Q$ *is a* transition *of* M *iff* $\langle o, q' \rangle = \delta(q, i)$*. We use* $T(M)$ *to represent the set of all transitions of* M*.*
2. *An* execution *of* M *is a finite sequence* $\langle q_0, i_0, o_0, q'_0 \rangle \cdots \langle q_{n-1}, i_{n-1}, o_{n-1}, q'_{n-1} \rangle \in T(M)^*$ *such that either* $n = 0$*, or* $n > 0$ *and the following hold.*
   Initiality. $q_0 \in Q_0$ *(the first transition starts from an initial state of* M*).*

Consecution. *For every consecutive pair of transitions* $\langle q_j, i_j, o_j, q'_j \rangle$ *and* $\langle q_{j+1}, i_{j+1}, o_{j+1}, q'_{j+1} \rangle$, $q'_j = q_{j+1}$ *(the target state of a transition is the source state of the following transition).*

We write $E(M)$ *for the set of executions of* M*.*

This papers deals with system invariants. For Mealy machines, the invariants we focus on are *transition specifications*, which we define (semantically) as follows.

Definition 3. *Let* $M = \langle Q, Q_0, I, O, \delta \rangle$ *be a Mealy machine.*

1. *A* transition specification ϕ *is any subset of* $Q \times I \times O \times Q$*.*
2. *Transition specification* ϕ *is* satisfied by transition $t \in T(M)$ *iff* $t \in \phi$*. It is* satisfied by execution $t_0 \cdots t_{n-1} \in E(M)$ *iff* t_i *satisfies* ϕ *for all* $0 \leq i < n$*.*
3. *Specification* ϕ *is an* invariant *of* M *iff each execution* $e \in E(M)$ *satisfies* ϕ*.*

Intuitively, a transition specification represents a desired relationship between source states, inputs, outputs and target states. A specification is satisfied by a transition if the transition is contained in the specification's set, and by an execution if every transition in the execution is so contained. A transition specification is then an invariant if it is satisfied by every execution.

2.2 Mealy Systems

Mealy machines may be seen as semantic objects. *Mealy systems* are instead more symbolic representations of Mealy-style behavior. In what follows, we fix a set $\mathbb{V}$ of *values* and use Y^X to represent the set of functions from X to Y.

Definition 4. *A* Mealy system *is a tuple* $\langle X, Q_0, U, C, \delta \rangle$*, where:*

1. $X = \{x_0, \ldots, x_{n-1}\}$ *is a finite set of* state variables*;*
2. $Q_0 \subseteq \mathbb{V}^X$ *is the (non-empty)* initial condition*;*
3. $U = \{u_0, \ldots, u_{m-1}\}$ *is a finite set of* input variables*;*
4. $C = \{c_0, \ldots, c_{l-1}\}$ *is a finite set of* output variables*, with* X, U *and* C *all pair-wise disjoint;*
5. $\delta \in (\mathbb{V}^X \times \mathbb{V}^U) \to (\mathbb{V}^C \times \mathbb{V}^X)$ *is the transition function.*

Intuitively, a Mealy system uses variables to represent states, inputs and outputs. Specifically, a state $q \in \mathbb{V}^X$ is a mapping from each state variable x_i to some value $q(x_i) \in \mathbb{V}$, while $i \in \mathbb{V}^U$ represents an input "vector" assigning a value $i(u_i) \in \mathbb{V}$ to input variable u_i. Outputs are treated similarly. The initial condition Q_0 explicitly specifies the initial system states, while δ similarly defines the transition function. In practice, Q_0 and δ are usually given symbolically; in addition, some state variables may also be seen as *parameters*, or variables whose values are not changed by any transition. In such a case, an initial condition would specify the allowed parameter settings.

Semantically, a Mealy system $M = \langle X, Q_0, U, C, \delta \rangle$ is interpreted as a Mealy machine $[\![M]\!] = \langle \mathbb{V}^X, Q_0, \mathbb{V}^I, \mathbb{V}^C, \delta \rangle$. The notions of transition and execution for Mealy machines (Definition 2) carry over directly to Mealy systems in the obvious manner. *Transition specifications* for Mealy systems will be first-order formulas defined over a Mealy system's variables.

Definition 5. *Let $M = \langle X, Q_0, U, C, \delta \rangle$ be a Mealy specification, and let $X' = \{x'_0, \ldots, x'_{n-1}\}$ be the set of* primed versions *of state variables; assume X' is pairwise disjoint from X, U and C.*

1. *A* transition specification *is any first-order formula ϕ whose free variables are drawn from X, U, C and X'.*
2. *Let $t = \langle q, i, o, q' \rangle \in T([\![M]\!])$ be a transition of $[\![M]\!]$. Then t* satisfies specification *ϕ, notation $t \models \phi$, iff $\rho_t \in \mathbb{V}^{X \cup U \cup C \cup X'}$ satisfies ϕ in the standard first-order sense, where ρ_t is given as follows.*

$$\rho_t(y) = \begin{cases} q(y) & \text{if } y \in X \\ i(y) & \text{if } y \in U \\ o(y) & \text{if } y \in C \\ q'(x) & \text{if } y \in X' \text{ and } y = x' \end{cases}$$

3. *ϕ is* satisfied by an execution $t_0 \ldots t_{n-1}$ *iff $t_i \models \phi$ for each i; it is an* invariant *of M iff $e \models \phi$ for each $e \in E(M)$.*

Intuitively, a transition specification for a Mealy system is a logical formula whose free variables refer to the state, input, output and primed state variables of a Mealy system. In the definition of $\models$, ρ_t is a valuation for these free variables; note that the source state of the transition is used to interpret the state variables in ϕ, while the target state defines the values for the primed state variables.

Example 1. To illustrate Mealy-system transition specifications, consider a simplified cruise-control system for an automobile. The state variables for the system include `status`, which indicates whether the system is off or on; `mode`, which indicates whether or not the cruise control is active, and thus controlling vehicle speed, or inactive, indicating that it is not; and `setSpeed`, which is real-valued and represents the speed the cruise control is trying to maintain. The inputs include: `switch`, which the driver can either set to on or off; `set`, a boolean value indicating whether or not the driver pressed the set-button on the control; `brake`, a boolean indicating whether or not the driver is stepping on the brake pedal; and `speed`, a real value indicating current vehicle speed. The only output, `throttleSetting`, is a real value in the range $[0, 1]$ indicating the percentage by which the throttle should be open. The set $\mathbb{V}$ of values is $\mathbb{R} \cup \{\mathsf{active}, \mathsf{false}, \mathsf{inactive}, \mathsf{off}, \mathsf{on}, \mathsf{true}\}$. Here are two sample invariants.

1. $(\texttt{status} = \mathsf{on} \wedge \texttt{brake}) \Rightarrow \texttt{mode}' = \mathsf{inactive}$. This asserts that if the driver steps on the brake and the cruise control is on, it must become inactive.
2. $(\texttt{status} = \mathsf{on} \wedge \texttt{set}) \Rightarrow (\texttt{setSpeed}' = \texttt{speed} \wedge \texttt{mode}' = \texttt{active})$. This says that if the cruise control is on and the driver presses the set-button, the set speed should be updated and the cruise control should become active.

We close this section by describing two notations for defining Mealy systems.

Discrete-Time Simulink. Simulink is a popular block-diagram notation for modeling control systems. The language has both continuous-time and discrete-time semantic accounts; the latter is the basis for modeling digital controllers, while the former is used for plant modeling and analog-controller design. A Simulink system consists of top-level inports and outports, and a series of interconnected blocks; the discrete semantics states that a block "fires", or evaluates, when all of its inputs are present. In addition some Simulink blocks store values. A discrete-time model executes a simulation step by first reading its inputs, fully evaluating blocks according to the evaluation rule just mentioned, which may have side effects including updating the values by some blocks, and producing outputs. Given this high-level account, it is immediate to see how discrete-time Simulink models give rise to Mealy systems: the data-store blocks correspond to state variables, the inports and outports define input and output variables, and the semantics of Simulink model evaluation define the transition function. The initial condition is implicitly defined by the Simulink semantics as well.

C Functions. In control-systems development a common approach to implementing a digital controller is to write a function in C that computes the desired control behavior. State information is encoded in variables defined externally to the function but read and written within the function; function inputs (if they are used) provide inputs, and the return value (if it is used) represent outputs.[1] Again, this gives rise naturally to a Mealy system: the external variables are the state variables, function inputs are the inputs, the return value is the output, and the function body defines how transition are computed.

3 Specstractor

Specstractor is the implementation of the specification-extraction framework originally presented in [1] and elaborated on in [5], and based on the Mealy-system formalism described in the previous section. It uses an iterative *test* → *infer* → *instrument* → *retest* cycle that uses data mining to extract proposed system invariants from sets of executions automatically generated from the system model. These invariants are then used to generate instrumentation for the model that checks whether the invariants are being maintained; the instrumented model has a new set of executions generated for it to double-check the validity of the proposed invariants and also to determine if new invariants can be proposed. Such executions may be seen as *tests*, as they include a sequence of inputs provided to the systems, and in what follows we will freely refer to them as such. In the remainder of the paper we will use Simulink models as our Mealy systems.

A key to the approach is the generation of "good" sets of executions. Ideally these executions should satisfy some coverage criterion; however, the traditional coverage criteria for state machines, such as state coverage and transition coverage, are not useful for the types of Simulink models we are interested in, since the

[1] In fact, it is common practice in these applications not to use C-function inputs and outputs, but instead use other external variables for this purpose.

number of states and transitions is too large to be effectively covered. Instead, we focus on coverage of the logic in the model, using adaptations of well-known structural-coverage metrics for source code (decision coverage, MC/DC, etc.) to Simulink. The Reactis®[2] tool supports the generation of tests with these types of coverage as goals; the resulting test cases, by exercising the boolean predicates in the model, provide some measure of semantic coverage of transition-computation aspects of model behavior without requiring coverage of the entire Mealy machine underlying the Simulink model.

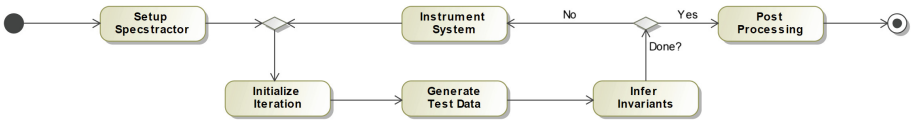


Fig. 1. Overview of the iterative control cycle of Specstractor

Figure 1 gives an overview of the Specstractor iterative cycle.

Setup Specstractor. This phase creates a user-selected folder for the execution of Specstractor and copies all necessary data to the folder (e.g. Simulink models). It also performs a one-time data-flow analysis of the system [5] to detect dependencies between inputs, state variables and outputs; this is used to ensure that generated invariants reflect causal relationships among these variables.

Initialize Iteration. Specstractor begins each iteration by creating a subfolder for the iteration's generated data (test cases, proposed invariants, etc.)

Generate Test Data. Test data is then created from the Simulink model using the Reactis test-generation tool, which uses a combination of Monte Carlo simulation, constraint solving and search heuristics to achieve high coverage of various structural metrics (e.g. decision coverage).

Infer Invariants. Once test cases are generated, Specstractor generates invariants in the form of *association rules*, which are implications whose left-hand sides (LHS) refer to the values of input and current-state variables, and whose right-hand sides (RHS) refer to output variables and new values of state variables. For this task, the tool uses a modified Frequent Pattern (FP)-Growth [6] algorithm from the Sequential Pattern Mining Framework (SPMF) [7]. The modifications to the FP-growth algorithm include the use of data from the static analysis to improve the performance of the algorithm [5] and the inclusion of the Range Miner algorithm (Sect. 4.1) to handle continuous variables.

Instrument System. To validate proposed invariants, the system models are instrumented with so-called *monitor models* [8]. Each monitor model is responsible for checking the truth or falsity of a single proposed invariant mined in

[2] Reactis® and Reactis for C® are registered trademarks of Reactive Systems, Inc.

the previous task. Reactis provides support for automating this instrumentation processes; Specstractor contains instrumenters that translate the mined invariants into the format that Reactis requires and automatically modifies the Reactis configuration file to integrate them into the test-generation process.

Retest. After the model is instrumented, we repeat the cycle in order retest the proposed invariants. The purpose of this retesting phase is to check whether the inferred invariants can be falsified with additional testing. Since the invariants are attached to the models as observers, they are treated as additional coverage objectives by Reactis, which then actively tries to find counterexamples to the proposed invariants. In the process it either disproves invariants by finding a test case that forces a violation of the invariants, or strengthens the confidence in them by creating additional tests that support them. In addition to validating existing invariants the additional test data can also uncover new behaviors which lead to invariants that were missed in previous iterations. For this reason, the test data of all previous iterations is aggregated with the newly created test data and a new set of invariants is inferred in each iteration.

Terminating the Process. The iterative process terminates if there no invariants are added to or removed from the set of invariants for n_t iterations, where n_t is a parameter set by the user.

Specstractor can be configured with different test generators, instrumenters and data miners, and thus can support Mealy-system notations besides Simulink.

4 Numerical Variables and Range Miner

Traditional association rule-mining algorithms such as Apriori [9] and FP-Growth [6] do not work well with unbounded data values, as they treat each value of an integer or floating-point number separately. For example, for one of our models, a cruise-control system of a car, the approaches would create the invariants speed = 24.4 ⇒ active = false, speed = 24.45 ⇒ active = false and so on. The validation step would then try to find counterexamples to each of these rules, creating more test data, and the subsequent data-mining step would find even more invariants. This stops the approach from converging, since it can always find invariants that differ from existing ones only in minor differences of the numerical constants appearing in the invariants.

If the user has pre-existing knowledge about the system, s/he can supply manually created abstractions for the data values that Specstractor would then automatically apply to the generated test data. However, the user would to be in possession of such information for this approach to be applicable.

Quantitative association-rule-mining [10,11] algorithms have been proposed to infer association rules from continuous variables; however, our preliminary experiments indicate performance and accuracy problems with them. We thus have created an approach, Range Miner, that builds on our existing iterative framework to infer accurate invariants over numerical variables.

4.1 Range Miner

The Range Miner algorithm uses the existing iterative process of Fig. 1 to refine an initial finite discretization of numerical variables (see Fig. 2(a)). It starts by analyzing the first set of test cases that are generated and discretizing the value range of any continuous variable using *binning* [12]. Invariants are then inferred from the binned data using the existing FP-Growth algorithm. Invariants with neighboring bins may also be merged [13] (see Fig. 2(b)) to reduce the number of invariants to manage.

Without manual abstractions each variable often can take many more different values (one for each bin), which leads to more false positives. We therefore supplement the test generator to better cope with the additional false positives. This is achieved with additional test data generated by fuzzing test cases (see Fig. 2(c)) from earlier iterations and by carrying over information for the test generator between iterations. After the process terminates each invariant with a binned variable is analyzed to check if its boundaries can be extended without violating the invariant (see Fig. 2(d)). The remainder of the section explains the steps in more detail.

4.2 Discretization via Fixed Point Binning

Discretization via data binning is a standard method to pre-process data for data-mining algorithms [12]. Data values are put into bins, or buckets, that represent a range of values. Two common types of binning methods are *equal length* and *frequency binning* (see 1a and 2a on the top left of Fig. 2). However, it is very unlikely that the automatically created bins line up completely with interesting points in the data (e.g. the 25 mph in our previous example). This can be rectified by the final invariant adjustment of the Range Miner; this is described below. However, by leveraging other characteristics of the Reactis test data, we can create a better initial binning that requires less adjustment. We call this modified method *fixpoint binning* (1b, 2b in Fig. 2).

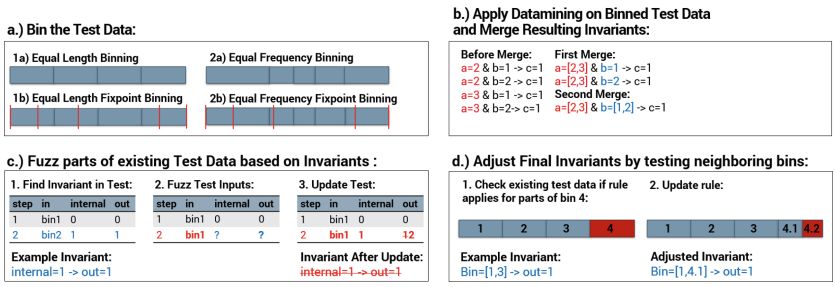


Fig. 2. Overview of the techniques used in the Range Miner approach.

A fixpoint is a bin with a single value that is created from additional information that we extract from the test data. Reactis produces two kinds of interesting

fixpoints that we take advantage of. Firstly, the most interesting values are often the most frequent ones in the test data. Secondly, floating-point values generated by Reactis rarely include a fractional component except when they are interesting points. If a fixed point is inside an existing bin the original bin is split. Even if the interesting points are not always accurate the approach will later rectify the situation by merging them together with neighboring bins.

4.3 Association Rule Mining and Merging of Invariants

The association-rule miner is unchanged apart from the fact that the test data is binned before it is handled by the miner. However, we add a step after invariant generation that performs an initial merging of invariants. This step is important for the performance of the test generator, which tends to deteriorate in the presence of too many invariants. The merging approach is illustrated on the top right of Fig. 2. It compares all the invariants of the same length and to determine if they are equal in all parts except for one variable; if they are then they are merged. For invariants involving more than two variables, finding the optimal merging strategy is a multi-dimensional knapsack problem [14]. We use a greedy algorithm to try to find the optimal merge strategy in these cases.

4.4 Improving the Test Generator

Compared to manual abstractions, Range Miner must deal with many more false positives. In order to better identify and remove these additional false positives we modify the existing test-generation process by adding three mechanisms: preloading of existing test data; fuzz testing; and the generation barrier.

In each iteration of Specstractor the Reactis test generator always starts at 0% coverage, carrying over no information from previous iterations. This is very inefficient because we already have existing test data that can cover many parts of the system. Reactis has a feature that allows the preloading of existing test data. However, if we load all the existing test data in each iteration Reactis would have to process more test data in each iteration, which would eventually damage its performance. Instead of preloading all data we sample the existing test data. In particular, we search the test data for the first n_p occurrences of an invariant, were n_p is a user supplied value. An invariant occurs in the test data when the test data makes both the LHS and RHS of an invariant true.

In order to invalidate an invariant the test generator must find a transition in a system execution in which the LHS of the invariant is true but the RHS is not. In deterministic systems this can only happen as follows. The system must be in a state where the LHS of the invariant is true and other inputs or state variables that are not part of the LHS change the output of the system in a way so that it makes the RHS of the invariant false. We therefore have incorporated fuzz testing to help with the invalidation of false positives (see Fig. 2). For each invariant the fuzz tester identifies n_f occurrences of the invariants in the test data (n_f is a user-tunable value). The test case is copied from the start of the test case to the test step where the invariant occurs. The fuzzer then modifies

input values that are not part of the LHS of the invariant to try to identify a counterexample to the invariant.

The generation barrier leverages an observation made during the evaluation of Range Miner. Specifically, we saw that the true invariants are almost always in the test data after two or three iterations. Any invariant appearing afterwards is nearly always a false positive. The generation barrier is a user-configurable limit that sets the number of iterations of the approach after which no new invariants are retained. After the barrier has been reached additional iterations will only try to remove the remaining false positives.

4.5 Invariant Extension

The bins that are created by the automated binning step are not always accurately lined up with interesting decision points in the system. For example in one of our models the mined invariant $\texttt{measuredValue} \in [130, 180) \wedge \texttt{timer} \in [5, 20] \Rightarrow \texttt{alarm} = 1$ states that if the measured value and the timer are in a certain range then the alarm is turned on. The invariant is correct but incomplete; the invariant describing the whole behavior of the system in this case would be $\texttt{measuredValue} \in [130, 180) \wedge \texttt{timer} \geq 5 \Rightarrow \texttt{alarm} = 1$. This invariant was not inferred since the maximum value the timer reached during the test execution was 20 before the timer was either reset or the test case ended. In order to combate this phenomenon we do a post-processing step after the invariant extraction has terminated to look for extensions to individual invariants.

In the post-processing step, we evaluate each invariant that contains a binned variable to check if any of the binned variables can be extended into neighboring bins, or even indefinitely. To achieve this we create hypothesized invariants and test them against the accumulated test data generated during the invariant extraction. If the invariant cannot be extended into the whole neighboring bin, we create other proposed invariants that sample the neighboring bin to determine if it is true for parts of the neighboring bin (see Fig. 2). The invariants that cannot be invalidated using existing data are rechecked using automated testing.

5 Experimental Evaluation

This section describes two experimental evaluations of Specstractor. In the first we apply Specstractor to 12 third-party models of automotive-control systems and medical devices and compare the resulting invariants with existing specifications for these models. We particularly study how the invariants may be used to identify issues in the requirements documentation and how deviations between the requirements and the system and inconsistencies in the requirements are manifest in the invariants. We also present effort data.

The second study evaluates the Range Miner algorithm. Specifically, we compare the execution time and resulting invariants of the Range Miner approach with invariants obtained using manually-developed abstractions. This study is performed on 5 models using continuous variables. The experimental setup is the

same as in [5], which describes in detail dependent and independent variables of the experimental approach and the data-collection mechanism.

Table 1 lists the 12 systems used in our evaluations: 11 model [1,5,8] the lighting- and cruise-control systems of a car, while one is a model of a blood-infusion pump [15]. The table shows the number of continuous variables in the model and the type/complexity of the specifications. Eight of the models have natural-language requirements descriptions ranging from two to five pages, for an overall total of 30 pages. Manual analysis of these yielded a total of 70 requirements (3–17 per model). Three of the models have specifications in the form of state machines. We counted each transition and state in a state machine as one requirement (4–6 states per model, with 9–16 transitions), for a total 54 requirements. We do not have any specifications for the Cruise Control system and it is therefore only used to evaluate the Range Miner.

Table 1. The systems used in the evaluations. The variables column shows how many inputs/state variables/outputs a system has, while the continuous column lists how many of these variables are continuously valued.

Name	Variables	Continuous	Blocks	Requirements	Requirement complexity
Cruise control	7/2/2	1/0/1	83	N/A	N/A
Emergency blinking 1	2/1/1	N/A	165	State machine	5 states/14 transitions
Emergency blinking 2	5/3/2	0/2/1	375	State machine	6 states/16 transitions
Emergency blinking 3	5/1/3	N/A	107	State machine	4 states/9 transitions
Daytime driving light	14/1/3	0/1/0	52	Natural language	5 pages/7 requirements
Fog light	10/3/4	0/1/0	59	Natural language	3.5 pages/12 requirements
High beam light	9/0/2	N/A	52	Natural language	3 pages/6 requirements
Low beam light	9/0/5	N/A	40	Natural language	4 pages/6 requirements
Parking light	7/0/7	N/A	83	Natural language	4 pages/6 requirements
Position light	9/0/7	N/A	48	Natural language	3 pages/5 requirements
Rear fog light	11/0/5	N/A	56	Natural language	4.5 pages/14 requirements
Blood pump	2/5/5	1/2/2	238	Natural language	3 pages/16 requirements

5.1 Comparison with Given Specifications

In this study we compared the Specstactor-derived invariants manually against given system specifications. We also recorded the effort that is necessary to map the resulting invariants to the specifications.

Table 2. Result of the mapping analysis.

Name	Invariants	Merged	False positive	Req.	Acc.	Inacc.	Undoc.	Not extracted	Expected to miss	Effort
Emergency blinking 1	46	46	0	14	14	0	7	0	0	90
Emergency blinking 2	49	29	1	16	12	4	0	0	0	60
Emergency blinking 3	65	43	0	9	9	0	0	0	0	60
Daytime driving light	31	30	0	7	3	3	0	0	1	40
Fog light	72	39	0	12	6	3	0	3	0	120
High beam light	42	21	0	6	1	5	0	0	0	45
Low beam light	48	12	0	6	0	5	0	0	1	40
Parking light	38	9	0	6	3	3	0	0	0	40
Position light	30	5	0	5	1	3	0	0	1	30
Rear fog light	66	28	0	14	6	6	0	0	2	90
Blood pump	39	28	1	16	6	7	0	0	3	80
SUM	526	290	2	111	76	39	7	3	8	695

Table 2 contains the following evaluation data for each model.

Name, Invariant, Merged, False Positive, Req. The name of the model is given, as is the number of invariants computed by Specstractor. For performance reasons we configured the data miner so that each RHS has only one conjunct; the number of such invariants computed by Specstractor is listed in the Invariants column. To compare the invariants against the requirements we then merged invariants with the same LHS into a single invariant with multiple conjuncts on the RHS. The number of such merged invariants is listed in the Merged Column. The False Positive column lists the number of invariants that are in fact not invariants, as determined by subsequent analysis. Using manual abstractions we did not observe any false positives for the evaluation systems. However, a few false positives remained while using the Range Miner algorithm; this issue is discussed in the next subsection. Req. contains the number of requirements for each model in the associated requirements specifications.

In the case of state-machine based specifications, this number coincides with the number of transitions in the specification, since each transition can be seen as an association rule that Specstractor should infer. This treatment of state machines stands in contrast to the automaton-learning framework found, for example, in LearnLib [16]: there, entire state machines, and not just transitions, are learned. In order to report the results of mapping invariants to these requirements, we created classifications that constitute the next columns in the table.

Accurate Specification. These are requirements that can be completely described by the extracted (merged) invariants. Note that in general, multiple invariants may be needed to describe a given requirement.

Inaccurate Specification. Such requirements are not accurately reflected in the extracted invariants. As an example the requirement specification for Blood Pump states that *"If Blood Pressure (CNAP) improves within 10 s of the silent warning alarm, then the system will clear the alarm, and resume blood infusion"*. The relevant invariant states that whenever the blood pressure improves the warning alarm is cleared, independent of the time.

Undocumented Behavior. This is system behavior that is not documented in the requirements specification but that is described by the extracted invariants. In the Emergency Blinking 1 model, for example undocumented behavior took the form of missing transitions in the state machine that was the requirements documentation for the system. With the help of the extracted invariants we were able to identify additional transitions that were implemented but not specified.

Not Extracted. These are behaviors of the system that are not covered by any invariant. In our study these took the form of invariants that had empty LHS, and thus were not detectable using the data-mining approach used in Specstractor.

Expected to Miss/Out of Scope. Any requirement that involves time or a system state that cannot be observed where the time/state cannot be observed would fall in this category.

Effort. The effort to analyze the resulting invariants varied between 30 and 120 min per model and on average was 63 min. These times included the comparison of the invariants against the requirements as well as a manual inspection of the system to determine whether an invariant describes an undocumented behavior or is a false positive. It does not include the computer time necessary to extract the invariants or the time to instrument the internal variables (10–15 min per model) so that the data miner could see the state values. For more information about the execution time of the approach we refer the reader our previous case study in [5], which contains a detailed execution time and memory consumption analysis for these models. The most important factor in the effort to apply the approach was the number and complexity of the resulting invariants and of the requirements. The most effort spent per model was on the *Fog Light* system, due to the fact that the Specstractor user missed instrumenting an unfamiliar Simulink block. Analyzing why this behavior was missed took about 50% of the analysis effort. Similarly, analyzing the undocumented

behavior in Emergency Blinking 1 was very time-consuming in comparison to checking the invariants against the specifications.

5.2 Range Miner Evaluation

We also evaluated the Range Miner (RM) algorithm against our manual abstraction (MA) approach. Table 3 shows the resulting invariants, the number of false positives, the average number of iterations and the average iteration time to perform the extraction (split into test generation, data mining and other) for both RM and MA. RM requires more iterations and more test-generation time per iteration and is therefore always slower than MA. This is due to the increased number of false positives that RM has to remove and the additional test data that we preload in the new test-generation process. The test-execution time tends to be the dominating factor in the Specstractor approach, and it is influenced mostly by the number of invariants to verify.

Table 3. Comparison between the manual abstraction and Range Miner algorithm

Name	Invariants		False positives		#Iterations		Iteration time (s)		Test time (s)		Data mining time (s)		Other time (s)	
	MA	RM	MA	RM	MA	RM	MA	RM	MA	RM	MA	RM	MA	RM
Cruise control	31	29	0	2	5	11	92	188	84	131	6	54	2	3
Emergency blinking 2	58	49	0	1	6	8	111	121	103	110	7	10	1	1
Daytime driving light	31	31	0	0	5	11	89	139	70	72	16	63	3	4
Fog light	72	70	0	0	5	9	114	170	109	160	2	3	3	7
Blood pump	39	30	0	1	5.2	12.2	122	166	109	124	11	39	2	3

The number of invariants differs between MA and RM because of the distinct ways that MA and RM partition the data. By merging values RM generated invariants that required two or more invariants to describe with MA. We did observe four false positives in RM that the test generator was unable to remove consistently; during the 10 repetitions of the experiments they appeared 6–7 times. Thus, the test generator can remove them, although not consistently.

The results show that we can accurately infer invariants over continuous variables, but in order to do so we need more compute time. However, it also shows that there are still some false positives that the test generator could not consistently remove.

6 Related Work

Accurate specifications are of critical importance in formal methods [17] but actual specifications of systems are often inaccurate and incomplete. Different specification-mining approaches have been proposed to address these issues [18].

Many approaches [19–21] infer state machines that are intended to represent a complete model of the system's behavior. Specstractor on the other hand focuses on deriving individual requirements in the form of association rules. Our reasoning is that these single requirements are easier to comprehend and compare against existing natural language specifications than inferred state machines. The Daikon tool presented by Ernst et al. [2] instruments a system under test and collects trace data from test cases supplied to Daikon. The generated trace data is then used to infer likely invariants. In contrast, we infer invariants from the test data directly. Furthermore our iterative approach verifies the inferred invariants via additional tests that try to find counterexamples, or improve our confidence in the mined invariants. Similar to Daikon, we designed Specstractor to be extensible so that we can support multiple languages and modeling notations in the future.

Quantitative association rule mining [10,11] has been proposed to mine association rules from numeric variables. Our experiments indicated however that the value ranges inferred were often inaccurate and the performance too slow for our purposes. The Range Miner on the other hand combines simple data binning with our existing iterative verification framework to refine the mined invariants. The GoldMine tool [22] uses decision trees to derive quantitative association rules. The application is the verification of hardware designs. However, by using single decision trees we often cannot infer rules that are true but have very low support and by using decision-forest-based methods (i.e. multiple decision trees) the rate of false positive rules is very high. Furthermore, decision trees do not offer the very useful property of association rules that invalid rules will not be inferred again if one piece of evidence contradicts them.

Zeng et al. [23] use mined invariants to minimize the size of a test suite. Specstrator instead leverages the mined invariants to create modified versions of the test suite to better invalidate false invariants.

7 Conclusion

This paper presents the Specstractor tool chain for the automated extraction and analysis of system invariants. It describes the implementation of the iterative framework that extracts invariants from automatically generated test cases using data-mining techniques. It also introduces the Range Miner algorithm, which allows the approach to automatically infer invariants involving numerical data. Finally, it presents an evaluation of the tool chain on 12 Simulink models and their associated specification artifacts. We demonstrate that the approach can accurately infer invariants that describe the system with few false positives, and that these invariants can be used to identify issues in the artifacts in the

form of mismatches between the specification and the implementation and missing specifications. The evaluation of the Range Miner shows that it can also accurately infer invariants from variables with continuous values. However, it requires more test-generation time and more iterations to invalidate additional false positives that do not appear when using manual abstractions.

For future work we are planing to add new types of invariants, including ones that span multiple transitions in a sequence as well as so-called event invariants that describe temporal relationships between events. Furthermore, we want to evaluate if model checkers can remove the false positives that we could not invalidate with the current approach, and we want to port the approach to C. We also plan to study issues related to scaling and larger models.

References

1. Ackermann, C., Cleaveland, R., Huang, S., Ray, A., Shelton, C., Latronico, E.: Automatic requirement extraction from test cases. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 1–15. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_1
2. Ernst, M.D., et al.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**, 35–45 (2007)
3. Cheng, X., Hsiao, M.S.: Simulation-directed invariant mining for software verification. In: Proceedings of DATE 2008, pp. 682–687. ACM, New York (2008)
4. Beschastnikh, I., et al.: Mining temporal invariants from partially ordered logs. In: SLAML 2011, pp. 3:1–3:10. ACM, New York (2011)
5. Schulze, C., Cleaveland, R.: Improving invariant mining via static analysis. ACM Trans. Embed. Comput. Syst. **16**, 167:1–167:20 (2017)
6. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: ACM SIGMOD Record, vol. 29, pp. 1–12. ACM (2000)
7. Fournier-Viger, P., Lin, J.C.-W., Gomariz, A., Gueniche, T., Soltani, A., Deng, Z., Lam, H.T.: The SPMF open-source data mining library version 2. In: Berendt, B., Bringmann, B., Fromont, É., Garriga, G., Miettinen, P., Tatti, N., Tresp, V. (eds.) ECML PKDD 2016. LNCS (LNAI), vol. 9853, pp. 36–40. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46131-1_8
8. Cleaveland, R., Smolka, S.A., Sims, S.T.: An instrumentation-based approach to controller model validation. In: Broy, M., Krüger, I.H., Meisinger, M. (eds.) ASWSD 2006. LNCS, vol. 4922, pp. 84–97. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70930-5_6
9. Agrawal, R., Srikant, R., et al.: Fast algorithms for mining association rules. Proc. VLDB **1215**, 487–499 (1994)
10. Srikant, R., Agrawal, R.: Mining quantitative association rules in large relational tables. In: Proceedings of SIGMOD, pp. 1–12. ACM, New York (1996)
11. Salleb-Aouissi, A., Vrain, C., Nortet, C.: Quantminer: a genetic algorithm for mining quantitative association rules. IJCAI **7**, 1035–1040 (2007)
12. Han, J., Pei, J., Kamber, M.: Data Mining: Concepts and Techniques. Elsevier, New York (2011)
13. Bay, S.D.: Multivariate discretization for set mining. Knowl. Inf. Syst. **3**(4), 491–512 (2001)

14. Kellerer, H., Pferschy, U., Pisinger, D.: Introduction to NP-completeness of knapsack problems. In: Knapsack Problems, pp. 483–493. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24777-7_16
15. Lindvall, M., et al.: Safety-focused security requirements elicitation for medical device software. In: Requirements Engineering, pp. 134–143. IEEE (2017)
16. Merten, M., Steffen, B., Howar, F., Margaria, T.: Next generation learnlib. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 220–223. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_18
17. Rozier, K.Y.: Specification: the biggest bottleneck in formal methods and autonomy. In: Blazy, S., Chechik, M. (eds.) VSTTE 2016. LNCS, vol. 9971, pp. 8–26. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48869-1_2
18. Zeller, A.: Specifications for free. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 2–12. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_2
19. Dallmeier, V., Knopp, N., Mallon, C., Hack, S., Zeller, A.: Generating test cases for specification mining. In: Proceedings of ISSTA, p. 85. ACM Press, July 2010
20. Le Goues, C., Weimer, W.: Specification mining with few false positives. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 292–306. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_26
21. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proceedings of ICSE, p. 501. ACM Press, May 2008
22. Vasudevan, S., et al.: GoldMine: automatic assertion generation using data mining and static analysis. In: Proceedings of DATE, pp. 626–629 (2010)
23. Zeng, F., Cao, Q., Mao, L., Chen, Z.: Test case generation based on invariant extraction. In: Proceedings of WCNMC, pp. 1–4. IEEE, September 2009

Bridging the Gap Between Informal Requirements and Formal Specifications Using Model Federation

Fahad Rafique Golra[1(✉)], Fabien Dagnat[2], Jeanine Souquières[1], Imen Sayar[1], and Sylvain Guerin[3]

[1] Université de Lorraine, CNRS, LORIA, 54000 Nancy, France
fahad-rafique.golra@univ-lorraine.fr
[2] IMT Atlantique, IRISA, Université Bretagne Loire, 29238 Brest, France
[3] Openflexo, 29280 Plouzané, France

Abstract. Software development projects seeking a high level of accuracy reach out to formal methods as early as the requirements engineering phase. However the client perspective of the future system is presented in an informal requirements document. The gap between the formal and informal approaches (and the artifacts used and produced by them) adds further complexity to an already rigorous task of software development. Our goal is to bridge this gap through a fine-grained level of traceability between the client-side informal requirements document to the developer-side formal specifications using a semi-formal modeling technique, model federation. Such a level of traceability can be exploited by the requirements engineering process for performing different actions that involve either or both these informal and formal artifacts. The effort and time consumed in developing such a level of traceability pays back in the later phases of a development project. For example, one can accurately narrow down the requirements responsible for an inconsistency in proof obligations during the analysis phase. We illustrate our approach using a running example from a landing gear system case study.

1 Introduction

General software development methods do not lend themselves to the kind of rigorous analysis necessary for ensuring the degree of assurance required for safety (or life) critical systems [1]. Formal methods attain that level of quality through proper documentation and significant analysis. In projects using formal methods, we usually come across domain artifacts (feasibility reports, existing models and software, standards, etc.), user requirements document (also serves as a frame of reference for the agreement between client and supplier) and the specifications document (used for formally defining the requirements). The specification documents are prepared to concretize the software development team's perspective of the software under development [2]. Where requirements document is an informal description of the system, a specification document uses rigorous formal methods that serve for verification and prototyping of a designed system.

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 54–69, 2018.
https://doi.org/10.1007/978-3-319-92970-5_4

As a development project progresses, artifacts contributing to its goal are produced. Traceability is the ability to link these artifacts together, so that one can identify the relationship between them and trace back/forward to them. Due to the gap between the informal and formal approaches, a requirement is taken as a single unit of reference for traceability [3]. This amounts to a coarse-grained traceability that overlooks individual concepts in each requirement [4]. We argue that a fine-grained level of traceability that can link individual concepts in formal specifications to the individual concepts of informal requirements shall help reduce this gap. Different approaches propose using a controlled natural language [5] to solve this issue. Even though the use of controlled natural language helps reduce requirements ambiguity, it hardly offers any support for improving the level of traceability between the requirements and specifications.

In a previous work, we proposed a concept-level traceability between the informal requirements and the formal specifications [6]. As an extension to that work, we use semi-formal models to formalize this traceability mechanism. We chose model federation [7] to realize a framework for the development and co-evolution of requirements and specifications. Model federation is an approach that enables binding a set of models from heterogeneous paradigms together. This binding is defined through a behavior that specifies the evolution of federated models in the development of a software system. Using this approach we developed an open source tool that can link requirements documented in various formats (word processors, spreadsheets, databases, xml files or Reqif supporting tools) to the formal specifications. We illustrate the use of our framework using examples from the landing gear system case study [8].

The rest of this paper is organized as follows. First, we describe the gap between the informal and formal approaches in Sect. 2. The model federation approach in presented in Sect. 3. In Sect. 4, we explain the structural core and in Sect. 5, we describe the methodological aspects of our framework. We share the lessons learned in various case studies, in Sect. 6. Then in Sect. 7 we discuss the state of the art. Finally, we conclude this paper in Sect. 8.

2 The Gap

Formal methods serve as the backbone of software engineering for critical and complex systems [9]. They guarantee the correctness of the system under development and help in early validation/verification of requirements. For example, Event-B [10] is a formal method for modeling and reasoning about large reactive and distributed systems. It is centered on the notion of transitions. Models are developed using two basic constructs: contexts and machines. Building a specification is a gradual process that uses context extension and machine refinement. Rodin [11], an Eclipse based IDE for Event-B provides effective support for refinement and mathematical proofs.

When working on a critical and complex system, one has to deal with both informal and formal models during requirements engineering and/or early architecture design. The most obvious informal model is the requirements document

that lists the requirements of the system to be built using natural language. Even though some approaches propose a controlled structure for writing the requirements [5], the requirements document remains informal. Tools like spreadsheets and word processors are still extensively used for maintaining the requirements [12]. Such tools along with other requirements management tools (*e.g.* Rational DOORS[1]) may provide the necessary flexibility for requirements management, but they offer little support when it comes to traceability. Other tools specifically designed for requirements traceability (*e.g.* Reqtify[2]) offer a very coarse-grained traceability. They consider a requirement as a unit concept and link it to other artifacts of software development. This makes it hard to keep track of individual concepts that form the core of a system design.

Unless the traceability approaches can pin down the concepts that lead to problems in specifications (*e.g.* conflicting requirements), the problem of gap between the informal and formal approaches in early software development can not be resolved. Imposing formal languages for documenting requirements is not possible because requirements elicitation is often a shared responsibility of clients and suppliers. In an earlier work, we also shared the case where the requirements document was almost completely prepared by the client organization [13]. Sometimes, the clients of critical and complex industry, especially from aviation and defense sectors, prepare the requirements documents in advance and then call for an open bidding for the projects. In requirement engineering, especially for critical and complex systems, where formal specification helps ensuring the correctness of the system, these informal requirements become the Achilles heel of the complete process. Coarse-grained traceability to the requirements level is not sufficient enough to cover this gap of informal to formal models. We argue that a very fine-grained level of forward and backward traceability to/from the specifications can reduce this gap. With such an approach at hand, one can even look forward to semi-automatic co-evolution of requirements and specifications.

3 Model Federation

Model Federation is a modeling approach where the focus is shifted from models to group of models. Instead of manipulating a single large model, it promotes using a set of interdependent models. This approach stems from the fact that a model is not an isolated entity, rather it depends on other models. For example one can federate a document file (.docx) with a list of states and a minimalistic state automaton (xml file format of UPPAAL). A model federation is therefore a chosen group of models reifying their dependencies to serve an intention. In the context of the automaton federation example, it aims at ensuring the consistency of the textual list of states and the automaton. An action on a member of a federation might impact the whole federation. Hence, each action on a model must be considered as an action on the federation. For instance, changing the name of a state is an action both on the document file and the automaton.

[1] https://www.ibm.com/us-en/marketplace/rational-doors.
[2] https://www.3ds.com/products-services/catia/products/reqtify/.

While this conceptual approach can be applied in any discipline using models, it is at its best when dealing with heterogeneous models pertaining to different paradigms. Co-evolution is difficult in such a scenario [14], especially if one wants to keep the various members of the federation in their respective paradigms [7]. It is often easier to act directly on the federation rather than acting on it members first and then recovering the consistency. The role of the federation therefore is to reify the process of ensuring consistency between the models of the federation. Notice that the level of consistency can vary, depending on the intention of its designer. Some federations may constrain the possible actions on the models of the federation to ensure a strict level of consistency. While others may choose to gather the inconsistencies and require human intervention to resolve them.

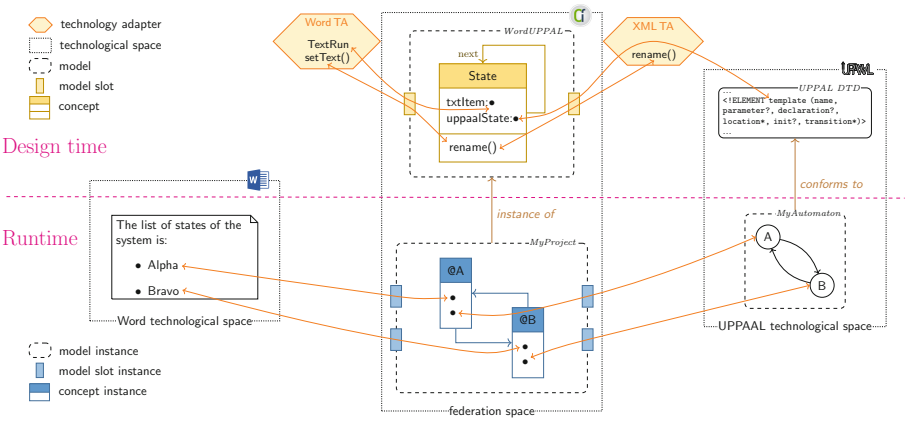


Fig. 1. An example of a model federation

The approach is relatively young but we have designed a framework with associated methods and tools. This framework relies on the following architecture. A federation gathers a set of conceptual models, named *virtual models* and a set of *federated models*. Each federated model pertains to a *technological space* and uses the language of its specific paradigm while a virtual model is built using the Federation Modeling Language (FML). Each federated model can be viewed as an autonomous component while the virtual models serve as control components. Figure 1 presents a simplified version of the automaton federation example that groups these different models. The upper half of the figure illustrates the design of a federation with a model coming from the Word technological space (.docx file), another coming from the XML technological space (UPPAAL file) and one virtual model reifying the dependencies between the textual data and the corresponding automaton. A *technology adapter* (TA) is a reusable library that defines connections between the FML execution engine and a particular technological space. The model federation framework provides ways to define TA. The *automaton* virtual model, shown in Fig. 1 relies on the two technol-

ogy adapters (Docx TA and XML TA) for accessing the models of respective technological spaces.

FML is a language designed to define virtual models. A virtual model is composed of a set of *concepts*, while itself being a concept. Hence, virtual model are structuring units while concepts are the core entities. A concept has a set of *roles* and *behaviors*. A parallel to object-oriented approach can be useful to understand FML[3]. A concept corresponds to a class, its roles to the attributes of the class and its behaviors to the methods of the class. In our example, a concept State is presented with two roles txtItem and uppaalState. These roles have types defining the kind of value the role will point to at runtime. Our example illustrates three forms of such types *i.e.* another concept for next, a type defined in a TA for txtItem or a type defined in an external model for uppaalState. Whenever a type external to the federation space (from a TA or an external model) is used, one needs to use a *model slot*. A model slot is a mediation entity, associated with a TA, in charge of giving access to external elements of the corresponding technological space. A model slot defines a view on an external model by interpreting it as a set external concepts. Notice that a model slot can limit its interpretation to the needed part of the external model.

FML is designed to define not only the structure of the virtual models but also their behavior. Beware, here the behavior means the collection of actions an engineer will be able to perform on a model federation. It is different from the behavior of the System Under Study. The rename operation already cited for the automaton federation is an example of this behavior. One could also define operations to add or remove states, transitions, *etc.* The renaming operation is defined by a rename behavior in the concept State. It uses the setText action of the Word TA and rename from the XML TA. When the FML execution engine runs a federation, it creates virtual model instances containing concept instances. Some concept instances are connected to external elements through model slot instances. The lower part of the Fig. 1 illustrates this runtime phase.

The tool support for model federation framework, Openflexo[4], is developed as an open source initiative with active community around it. This tool offers a FML execution engine with an interactive virtual model design environment. It has been used in several use-cases including model mapping, multi-paradigm process modeling and enterprise architecting. It has also been used to build a tool, the freemodeling editor that has been put to practice in industrial projects [15]. As of today, this tool offers some mature technology adapters (*docx* and *excel* for documents, EMF and OWL for modeling languages, JDBC for databases, REST and XMLRPC for external services and one for diagramming tools) and some other rudimentary ones (pdf, http, XML and powerpoint).

[3] Beware, even though useful for comprehension, this correspondence is not reliable, as some aspects of FML do not map to object oriented concepts.

[4] http://openflexo.org and https://github.com/openflexo-team.

4 Linking Requirements to Specifications

In order to overcome the informal to formal barrier, as described in Sect. 2, we developed an approach to link requirements and formal specifications using a fine-grained level of traceability. We develop a model federation using three virtual models in federation space *i.e.* requirements, specification and glossary virtual models. For the requirements, this federation connects to any of the three technological spaces shown in Fig. 2. It may connect to other technological spaces for which the tool offers a technology adapter *e.g.* databases, EMF, service oriented platforms, *etc.* For formal specifications, currently we are supporting Event-B technological space through XML technology adapter.

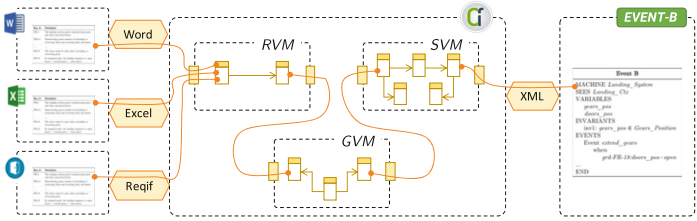


Fig. 2. Model federation for requirements to specification tracing

4.1 Requirement Virtual Model

We benefit from the strength of model federation by developing a virtual model for requirements. Instead of rewriting the requirements using a formal grammar or transforming the requirements into another (formal or semi-formal) model, the *requirement virtual model* interprets the textual requirements for our specific use of traceability. It allows identifying and tracing back to individual concepts within the textual description. For our specific use, we only need two concepts from a requirement *i.e.* the requirement identifier and its textual description. Functional and non-functional requirements are treated the same way, as long as there is a corresponding concept in both the requirement and the specification. Using existing *Technology Adapters*, requirement virtual model can connect to heterogeneous platforms to get any requirements from a word processor, spreadsheet or RIF/ReqIF compliant XML formats (*e.g.* DOORS, ProR, etc.).

`Requirement` concept of requirement virtual model, as illustrated in Fig. 3 gets the identifier and the textual description of the requirement from the connected *Requirements Technological Space*. It also contains two boolean type roles `isValidated` and `isConsidered` that are used to relay back the information to the concerned stakeholders about the status of a requirement; whether or not it was included in the specification and does it pose any proof obligation issue. The `IdentifiedConcept` refers to a word/sub-phrase in the textual description of the requirements document which carries an equivalent formal concept

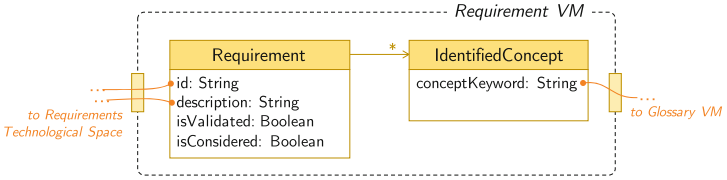


Fig. 3. Simplified requirement virtual model

in the system specification. The exploration and selection of concepts ensuring the conformity to the defined needs is a progressive activity carried out for the development of specifications in complex software development. For example, the European Cooperation for Space Standardization details a complete process in *ECSS-E-ST-10-06C*, starting from the identification of possible concepts to the establishment of technical specifications [16]. The behavior part of this model, not shown in the figure, allows to describe the operations like relaying the information to the stakeholders, observing any change in the requirements, triggering certain behavior in the connected Glossary VM, etc.

4.2 Specification Virtual Model

Specification VM interprets a specification with the intention of linking it to the requirements. The long-term objective of this virtual model is to interpret specifications from different (lightweight) formal methods, however we have focused on the Event-B specifications for the moment. Still, the use of model federation offers tool independence by allowing us to use the same virtual model for *Atelier B*, *Rodin* or *B-Toolkit*. Figure 4 illustrates the key concepts of the specification virtual model. The notions of local variable, action, variants and invariants, *etc.* are abstracted behind `EventProperty` and `MachineProperty` for brevity.

The goal of interpreting an Event-B model is to gather the formal concepts from a specification and to identify the kind of those concepts. A formal concept can be of any kind *e.g.* a machine, variable, constant, etc. We believe that

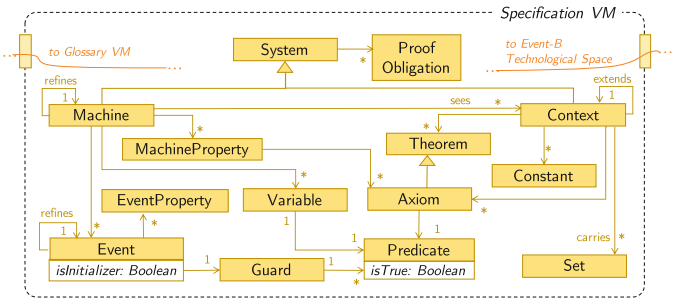


Fig. 4. Excerpt from the specification virtual model

the behavior of a trace link between an informal concept and a formal concept depends on the kinds of those concepts. For example, when an informal concept in a requirement is traced to a constant concept of a specification, one can define the behavior of the trace link such that a change in an informal concept automatically changes the formal concept, but a change in formal concept requires a validation of a requirement engineer to be propagated to the informal concept. For such federation level behavior, the user can add an operation in Specification VM that triggers the behavior of the linked virtual models. Specification VM already contains the behavior for observing the Event-B technological Space and triggering various behaviors of the Glossary VM.

4.3 Glossary Virtual Model

Glossary VM is the key model that binds *Requirement VM* and *Specification VM*. The `InformalConcept` from this model is linked to the Requirement VM, from where it gets the informal keyword of the concept. This keyword can be of any kind specified by the `InformalCKind`. For the moment, we have left it to the user to choose the kind of the concept, but for an industrial application, one can integrate Natural Language Processing techniques (*e.g.* [17]) that can parse, extract and categorize the concepts from the requirement descriptions (Fig. 5).

The `FormalConcept` of the glossary virtual model is linked to the *Specification VM*, from where it gets the `formalKeyword`. It also specifies the kind of the concept amongst the ones described by the `FormalCKind`. Where `InformalConcept` only defines the basic manipulation behaviors like `create`, `update` and `delete`, the `FormalConcept` also defines the `refine` and `isPOTrue` behaviors. The `refine` behavior carries the information about the refinement of an Event-B machine to the corresponding requirement in the requirements document. The `isPOTrue` behavior notifies the concerned stakeholders of the requirement, whether or not the proof obligations of the corresponding Event-B machine are validated. `Trace` binds the `InformalConcept` with the `FormalConcept`. Each *trace* has a unique identifier. The `create` and `update` behaviors assign a kind to the trace amongst the ones defined by `TraceKind`:

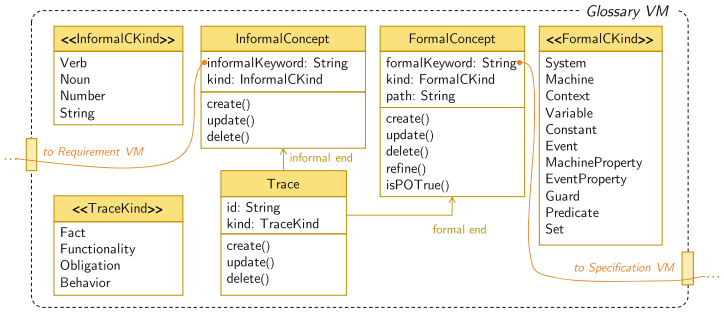


Fig. 5. Glossary virtual model

- A **Fact** connects an informal concept of `Noun`, `Number` or `String` kind to a formal concept of `Constant`, `Set`, `Variable`, etc. These trace links are used by requirements that specify facts about the future system.
- A **Functionality** connects an informal concept of `Verb` kind to a formal concept of `Event` kind. A requirement linked to this kind of trace describes an expected functionality of the system under development.
- An **Obligation** connects an informal concept of `String` kind to a formal concept of `Axiom`, `Invariant`, `Theorem` and `Guard` kinds. These trace links are used for preconditions or postconditions on the functioning of a system.
- A **Behavior** is a collection of *Functionality* links, such that their events must be performed in a temporal order. The informal concept is a `String` (a sub phrase) that contains the functionality concepts along with their order and links them to each of their corresponding formal concepts. The informal concept in the requirement specifies a behavior expected of the future system.

It is important to note here that our objective is not to exhaustively define the behavior of all kinds of trace links, but to propose a mechanism where such behavior can be specified. The three virtual models *i.e.* RVM, SVM and GVM, are used by the tool providers. An end user defines its traceability links with a set of available behavioral templates provided by the tool provider. The virtual models are instantiated in the background. Unless the user needs a customized behavior, she does not need to work with the virtual models. A strong motivation for choosing model federation was its ability to decouple the technological dependence from the core behavior of the methodology. For example, the behavior chosen for a trace link between a requirement and Event-B machine does not depend on the tool used for documenting the requirements. The technological aspects of a requirement coming from DOORS or from an Excel spreadsheet do not alter this behavior. They are confined in the Technology Adapter.

5 Process Driven Approach

To witness the full potential of our framework, one needs to define the process that integrates the building blocks from the models presented in the previous section. Using the Openflexo tool, we have developed a process modeling editor that serves for defining the process, as shown in Fig. 6. Once defined, each activity is linked with the behaviors of individual virtual models for the execution. The user has the liberty to define a desired behavior for each activity.

As the tool follows a process driven approach, it does not impose a specific behavior to the user. We demonstrate the use of our framework by discussing few examples of the activities from the landing gear system case study [8].

1. *Initializing a machine:* At the start of the specification process, there exists no corresponding Event-B machine for the concepts identified in the requirements document. The user identifies an informal concept, for which (s)he plans to develop an Event-B machine. Same thing happens, when a concept found in the requirements document does not correspond to any existing

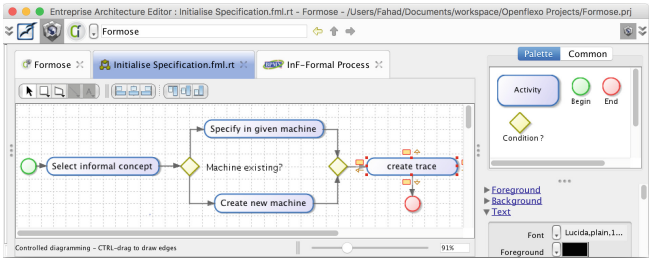


Fig. 6. Enterprise architecture editor

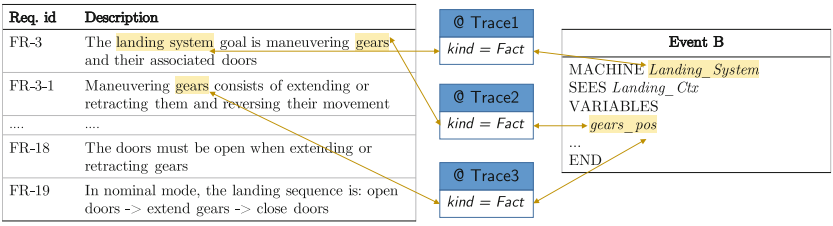


Fig. 7. Virtual model instance - initialization (This figure does not show the instances of RVM and SVM for the reason of brevity.)

machine and the development of a new machine is intended. In the example of Fig. 7, the user identifies "landing system" as informal concept. As per the process of Fig. 6, when no corresponding machine exists, the framework offers to create a new one. *Specification VM* can be used for creating a stub for the new machine. Once the machine is created, "*Landing_system*" is identified as a machine concept of *Specification VM* and it is linked to the informal concept "landing system".

2. *Updating a trace link:* The default behavior of the framework when the name of an Event-B machine (*e.g. Landing_System*) changes is to update the glossary and maintain the trace link. Conversely, when the name of the informal concept *landing system* changes, the framework generates a notification for the system analyst to check for probable inconsistencies. Even though the framework provides a default behavior for the activities associated with tracing, thanks to the model federation approach, the user has the flexibility to define a different behavior. Usually the behavior of such activities is encoded within the tool implementations, and changing the behavior is impossible or at least difficult in case of open source software. For model federation, this behavior is not coded in the tool, rather defined in the model itself.
3. *Adding new trace links:* The process of linking requirements to formal specifications is fairly flexible and varies from case to case. One can choose an informal concept and link it to an existing formal concept. But it can also go the other way round, when one selects an existing formal concept and links it to a concept from a requirement. In Fig. 7, we see that "gears" from FR-3

is linked to the "*gears_pos*" in *Landing_System* machine. Once this link is formed, the user goes in the reverse direction for linking this formal concept to other identified concepts from the requirements that refer to it. Similarly, the user links the informal concepts "extending" and "extend gears" to a corresponding formal concept "*extend_gears*" through a trace link of type functionality, as shown in Fig. 8. It is important to know that multiple derivatives of a single informal concept (*e.g.* extending, extend gears, etc.) do not need to have different corresponding formal concepts. One can notice in the figure that the complete description of FR-18 forms an informal concept linked to the guard of the *Landing_System* machine.

4. *Early requirements validation:* The process of tracing the requirements to the formal specifications forms the basis of validation. A requirement can have trace links to multiple Event-B machines and similarly a machine can have links to multiple requirements. The process of validation does not need to wait till the requirements document is complete. Hence, as the requirement document is under development, the formal specification process can start in parallel. This way requirements are traced to the specifications as and when they arrive. The `isPOTrue` property of the formal concept in the *Glossary VM* keeps track of the proof obligations of the linked Event-B machine. If the proof obligations of all the machines traced by a requirement's concepts are proved, the `isValidated` property of the requirement virtual model returns *true*. If a requirement is linked to the glossary but the trace is not complete to the formal specification, the `isConsidered` property of the requirement virtual model return *false*. This way the information about the validity of the requirements is relayed back to the stakeholders.
5. *Refinement of requirements and specifications:* Requirement elicitation is a gradual process that involves the refinement of abstract level requirements to the concrete level requirements. This process can be carried out at requirements or specifications. When an informal requirement is refined, all the informal concepts reappearing in the refined requirements with their corresponding formal concepts are notified to the stakeholder for possibly new trace links. In case of a refinement of an Event-B machine, the information is passed

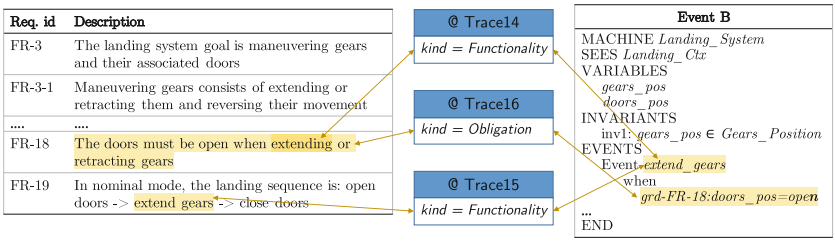


Fig. 8. Virtual model instance - traces (This figure does not show the instances of RVM and SVM for the reason of brevity.)

through the `refine` property of the glossary virtual model. The goal is to trigger new trace links from the requirements to the new machine.

The tool implementation[5] for this approach also supports requirements elicitation, links to domain models, ontologies and goal models, *etc.* but they are out of scope for this article. The intention is to show parts of the default process and the possibility to define user-specific processes. The tool being modular, we have linked this *requirement to specification traceability module* with our previous modules e.g. goal modeling and requirements elicitation modules [13].

6 Lessons Learned

During the early development of the approach, we focused on two case studies *i.e.* the landing gear system [8] and the hemodialysis machine [18]. Finally we implemented our approach on a real-life case study provided by our industrial partner from aviation and aerospace industry, under a research project, FORMOSE, funded by French National Research Agency (ANR)[6]. Based on our experiences with these implementations, we share the following lessons learned:

1. *Parallel incremental development of requirements and specifications:* During our case studies, we found out that it is not necessary to wait for the requirements document to start the specifications. As soon as requirements start to pour in, the development of specifications can start. The parallel and incremental development of both these artifacts helps requirements elicitation.
2. *Fine-grained traceability:* We applied a very fine-grained level of traceability going down to the concept level, within each requirement. Such a level of traceability helped the analysts to associate proper requirements to the justifications of each concept at the specification level. It also helped in categorizing the requirements according to their corresponding implementations in the formal specifications. Prioritization of requirements from the clients perspective is a common practice, but when combined with the specification view of priorities, it helps stakeholders to take informed decisions.
3. *The validation of the informal requirements:* Late validation of requirements after the requirements engineering phase increases the cost of corrective measures. The proposed framework helped in bringing the validation process close to requirements elicitation so much that the validation of requirements is done in parallel with the requirements document development. Once developed, it helps in proving the correctness of the formal specification in relation to the concepts present in the informal requirements, all along the development process. Maintaining the trace links in a glossary reduces the effort of validating complex specification models through refinement.
4. *Automation of traces:* One of the main reasons for using model federation for the linking informal requirements to formal specifications was the possibility

[5] FORMOD tool is available at https://downloads.openflexo.org/Formose.
[6] Bound by a non-disclosure agreement, we can't share the details of this case study.

of operationalizing the trace links. A trace link is not just a pointer from an informal concept to a formal concept, it contains a behavior. This allows to (semi-)automate some tasks of linking the two artifacts. Some behaviors we came across during the case studies were automatic update of constants, triggering notifications to the stakeholders, generating requests for proof obligations, changing the state of requirements from valid to conflicting, etc.

5. *Verification of requirements:* Because the traceability of requirements to formal specification was taken to a fine-grained level, the introduction of corresponding concepts helped in the verification of formal specifications. We found out that such traceability helps in detecting omissions in the requirements (initial states, implicit undescribed requirements, or absence of scenarios) or contradictions between the specifications and requirements.
6. *Specification versions of clients documents:* An interesting finding of the implementation of our framework to the industrial project was that the specification stakeholders relate more with the formal text than the informal one. The available trace links made it very easy to generate a version of requirements document where the informal concepts were replaced with the formal ones, as shown in Fig. 9. This eased their comprehension of requirements and improved the communication within the team.

Req. id	Description
FR-3	The [Landing_System] goal is maneuvering [gears_pos] and their associated [doors_pos]
FR-3-1	Maneuvering [gears_pos] consists of [extend_gears] or [retract_gears] and reversing their movement
...	...
FR-18	The [doors_pos] must be [open_doors] when [extend_gears] or [retract_gears]
FR-19	In nominal mode, the landing sequence is: [open_doors] → [extend_gears] → [close_doors]

Fig. 9. Specification version of requirements document

7 Related Work

The gap between the informal requirements and the formal specifications was acknowledged and has been a topic of research interest for the past three decades. Deriving VDM specifications from Structural Analysis (mostly Data Flow Diagrams) [19], Object Constraint Language (OCL) specifications from UML use cases [3] and system specifications from Problem Frame descriptions [4] are few of the examples to bridge this gap. However, most of such efforts are tools and technology specific endeavors. They lack a generic methodology that can be applied in a variety of configurations. The main focus of our work is to reduce the technology dependence from the methodology of linking informal requirements to formal specifications. From the requirements perspective, we accept requirements coming from any tool or described in any formalism. For the specifications, we only implemented Event-B model for now. This is an implementation shortcoming but it does not alter the proposed methodology.

KAOS is a refinement-based goal-oriented methodology for deriving specifications formalized using Linear Temporal Logic from informal requirements [20]. KAOS facilitates the derivation of system specifications through refinements, but imposes its own requirements description (goal modeling) methodology and constrains the language used for formal specifications. Besides, KAOS suffers from the lack of support for non-functional requirements. Li et al. [21] present another refinement based approach for the transformation of informal requirements to formal specifications, that can handle NFRs. They use a requirement ontology for classification and propose a requirements modeling language. The advantage of the proposed methodology is that the user is not restricted to any specific requirements specification method or tool.

Our work is notably closest to the approach of Jastram et al. where their requirement model differentiates between phenomena (state space and transitions of the system) and artifacts (the restriction on states and transitions) [22]. They classify the artifacts into Domain Knowledge (W), Requirements (R), Specifications (S), Program (P) and Programming Platform (M). Once formalized, these elements of requirements are mapped to Event-B, using ProR [23]. The main difference with our approach is that we propose explicit definition of traceability behavior in the trace links. Apart from mapping the concepts of informal requirements to formal specifications, we classify different kinds of trace links. The behavior of each kind of trace link is then reused for providing semi-automatic co-evolution of requirements and formal specifications.

Heisel et al. proposed a requirements elicitation process that is independent of the specification language [24]. An extension to their work using Rodin and ProR offers a fine-grained traceability between informal requirements and formal specifications [6]. We have proposed yet another extension to this work using model federation. The advantage of this extension is that the trace links are now fine-grained to a concept level and that they contain the behavior of the trace, rendering them executable. The main limitation of our approach is the cost it incurs. Indeed, the process of maintaining the consistency between the requirements and the formal specifications must be reified. A method engineer has to define the corresponding behavior. Furthermore, the specification engineer needs to invest time for linking individual concepts of informal requirements to the formal specifications. However, this investment provides improved clarity and the possibility of automating certain activities in case of requirements change. Notice also that probably common behaviors would emerge and provide a set of reusable federation behaviors, lowering the cost of their design.

8 Conclusion and Future Work

We proposed a framework for linking informal requirements to formal requirements specifications. The main contributions of our approach are: (i) fine-grained traceability between individual concepts in a requirement and individual concepts in a formal specification, (ii) a mechanism for the incorporation of behavior within the trace links, and (iii) tooling and methodology for the incremental

development and co-evolution of requirements and formal specifications. We are currently working on an operational semantics of FML using a process calculus encoding similar to [25]. The aim of this semantics is twofold. One, we plan to check the correctness of our FML interpreter. Second, we would be able to prove properties about behaviors such as, for example, any modification of an artifact leads to a corresponding modification of depending artifacts. Proving properties of the process alongside the product would help certification of critical systems.

References

1. Coram, M., Bohner, S.: The impact of agile methods on software project management. In: 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, ECBS, pp. 363–370. IEEE (2005)
2. Clark, R.G., Moreira, A.M.: Formal specifications of user requirements. Autom. Softw. Eng. **6**(3), 217–232 (1999)
3. Giese, M., Heldal, R.: From informal to formal specifications in UML. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 197–211. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30187-5_15
4. Seater, R., Jackson, D., Gheyi, R.: Requirement progression in problem frames: deriving specifications from requirements. Requir. Eng. **12**(2), 77–102 (2007)
5. Mavin, A., Wilkinson, P., Harwood, A., Novak, M.: Easy approach to requirements syntax (EARS). In: International Requirements Engineering Conference, pp. 317–322. IEEE (2009)
6. Sayar, I., Souquières, J.: La validation dans les premires étapes du processus de développement. ISI-DAT **22**(4), 11–41 (2017)
7. Golra, F.R., Beugnard, A., Dagnat, F., Guerin, S., Guychard, C.: Addressing modularity for heterogeneous multi-model systems using model federation. In: Companion Proceedings of the International Conference on Modularity, pp. 206–211. ACM (2016)
8. Boniol, F., Wiels, V.: The landing gear system case study. In: Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. CCIS, vol. 433, pp. 1–18. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07512-9_1
9. Rierson, L.: Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance. CRC Press, Boca Raton (2017)
10. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
11. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int. J. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
12. Behutiye, W., Karhapää, P., Costal, D., Oivo, M., Franch, X.: Non-functional requirements documentation in agile software development: challenges and solution proposal. In: Felderer, M., Méndez Fernández, D., Turhan, B., Kalinowski, M., Sarro, F., Winkler, D. (eds.) PROFES 2017. LNCS, vol. 10611, pp. 515–522. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69926-4_41
13. Golra, F.R., Beugnard, A., Dagnat, F., Guerin, S., Guychard, C.: Continuous requirements engineering using model federation. In: 24th International Requirements Engineering Conference (RE), pp. 347–352, September 2016

14. Hebig, R., Giese, H., Stallmann, F., Seibel, A.: On the complex nature of MDE evolution. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 436–453. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41533-3_27
15. Golra, F.R., Beugnard, A., Dagnat, F., Guerin, S., Guychard, C.: Using free modeling as an agile method for developing domain specific modeling languages. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS 2016, pp. 24–34. ACM, New York (2016)
16. ECSS: Space Engineering - Technical Requirements Specification. Standard ECSS-E-ST-10-06C, European Cooperation for Space Standardization (2009)
17. Arora, C., Sabetzadeh, M., Briand, L., Zimmer, F.: Automated checking of conformance to requirements templates using natural language processing. IEEE Trans. Softw. Eng. **41**(10), 944–968 (2015)
18. Mashkoor, A.: The hemodialysis machine case study. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 329–343. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_29
19. Fraser, M.D., Kumar, K., Vaishnavi, V.K.: Informal and formal requirements specification languages: bridging the gap. IEEE Trans. Softw. Eng. **17**(5), 454–466 (1991)
20. Van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software, vol. 10. Wiley, Chichester (2009)
21. Li, F.-L., Horkoff, J., Borgida, A., Guizzardi, G., Liu, L., Mylopoulos, J.: From stakeholder requirements to formal specifications through refinement. In: Fricker, S.A., Schneider, K. (eds.) REFSQ 2015. LNCS, vol. 9013, pp. 164–180. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16101-3_11
22. Jastram, M., Hallerstede, S., Leuschel, M., Russo, A.G.: An approach of requirements tracing in formal refinement. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 97–111. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15057-9_7
23. Jastram, M.: ProR, an open source platform for requirements engineering based RIF. In: Systems Engineering Infrastructure Conference, SEISCONF (2010)
24. Heisel, M., Souquières, J.: A method for requirements elicitation and formal specification. In: Akoka, J., Bouzeghoub, M., Comyn-Wattiau, I., Métais, E. (eds.) ER 1999. LNCS, vol. 1728, pp. 309–325. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-47866-3_21
25. Wong, P.Y., Gibbons, J.: Formalisations and applications of BPMN. Sci. Comput. Program. **76**(8), 633–650 (2011)

Concurrency

Program Verification for Exception Handling on Active Objects Using Futures

Crystal Chang Din[1(✉)], Rudolf Schlatte[1], and Tzu-Chun Chen[2]

[1] Department of Informatics, University of Oslo, Oslo, Norway
{crystald,rudi}@ifi.uio.no
[2] Department of Computer Science, Technische Universität Darmstadt, Darmstadt, Germany
tc.chen@dsp.tu-darmstadt.de

Abstract. For implementing correct systems, handling and recovering from exceptional situations is important but challenging for ensuring correct interactions among distributed objects which are processing concurrently. To focus on exploring novel handling constructs for actor-based programming languages, we study ABS, an actor-based concurrent modeling language with an underlying executable formal semantics. This paper introduces multi-party session blocks with recovery handlers for exceptions into ABS. With this novel construct, we verify the correctness of interactions among objects within a session block. Program correctness is ensured by specifying invariants as pre- and post-conditions, called session contracts, for such a block, which is more expressive than the existing class invariant proof system for ABS. We present the extension of ABS with a try-catch-finally construct and class session recovery blocks that handle uncaught exceptions.

1 Introduction

Properly handling and recovering from exceptional situations is an important part of specifying and implementing robust and correct systems, especially for distributed systems where correctness must take partial failure scenarios into account [17]. Therefore, modeling languages should include means of specifying exceptional situations and how to recover from them. This paper presents a new approach to expressing multi-party exception transmission and recovery for active object languages [4]. We designed the approach for the modeling language ABS [13]. This paper adds standard language constructs to specify, raise and handle exceptional situations, as well as a novel construct, the *session block*, for reestablishing object invariants after unhandled exceptions.

Existing class invariant-based proof theories for ABS [7] are restricted in expressivity, specifically in the area of upholding guarantees of protocols involving series of message exchanges between multiple participants. The problem is

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 73–88, 2018.
https://doi.org/10.1007/978-3-319-92970-5_5

that the semantics of ABS process interleaving and scheduling cannot forbid arbitrary messages to be processed in-between the expected ones, requiring whole-program analysis. This paper addresses this problem by introducing the concept of *sessions*, which temporarily restrict the scheduling behavior to the parts of a model participating in the session. In this work, we define *session contracts* to express the desired properties of a session based on the new session construct. A proof system for session contracts is introduced.

The rest of the paper is structured as follows. Section 2 describes the main characteristics of the ABS language. Section 3 introduces the new language constructs. Section 4 introduces session contracts and provides a proof system for verifying session contracts. Section 5 discusses related work. Section 6 discusses future work and concludes the paper.

2 A Short Introduction to ABS

The ABS language was developed to model distributed, parallel systems. Its design makes it amenable to both formal analysis and simulation (execution). The syntax is similar to languages in the C/Java family tree. ABS is an actor-based active object language, with interface inheritance and code reuse via traits. Being an active object language means that objects are "heavy-weight": method calls create processes on the target object, which are scheduled cooperatively in each concurrent object group (cog). Process switching occurs only when the current process terminates or at clearly marked program locations (*await* statements); this makes models of concurrent and distributed systems amenable to compositional analysis and proof. Data is modeled via a functional sub-language consisting of algebraic datatype definitions and side effect-free functions.

2.1 A Brief Example

Figure 1 shows a complete ABS model simulating bank accounts and transactions involving multiple accounts. The `Account` interface and `CAccount` class model a bank account with the usual deposit, withdrawal and balance inspection methods. Methods of type `Unit`, e.g., `deposit`, can omit an explicit `return Unit;` statement. The `Transaction` interface and `CTransaction` class model the control flow that models a transaction involving transferring some funds from one account to another, with a small commission transferred to a third account. The method `transfer` of the `Transaction` class (Line 17) first deducts the given amount from the sender account, then calculates the commission and deposits the proper amounts in the receiver and commission accounts.[1]

ABS object references are typed via interfaces, which describe the set of messages that an object can process (lines 3, 8). Classes (lines 11, 16) implement zero or more interfaces and contain method definitions. Method calls (e.g., Line 20) are *asynchronous*, written `o!m()`, and create a new process in the callee.

[1] The slightly awkward calculation of `profit` is used to introduce a runtime error.

```
1  module BankAccount;
2
3  interface Account {
4      Unit deposit (Rat amount);
5      Unit withdraw (Rat amount);
6      Rat getBalance();
7  }
8  interface Transaction {
9      Unit transfer (Account f, Account t, Rat amount);
10 }
11 class CAccount(Rat balance) implements Account {
12     Unit deposit (Rat amount) { balance = balance + amount; }
13     Unit withdraw (Rat amount) { balance = balance - amount; }
14     Rat getBalance() { return balance; }
15 }
16 class CTransaction(Account commission, Rat factor) implements Transaction {
17     Unit transfer (Account sender, Account receiver, Rat amount) {
18         await sender!withdraw(amount);
19         Rat profit = amount / factor;
20         commission!deposit(profit);
21         receiver!deposit(amount - profit);
22     }
23 }
24 {
25     Account f = new CAccount(50);
26     Account t = new CAccount(50);
27     Account c = new CAccount(0);
28     Transaction trans = new CTransaction(c, 10);
29     await trans!transfer(f, t, 10);
30     Fut<Rat> fp = c!getBalance();
31     await fp?;
32     Rat profit = fp.get;
33     println("Profit: " + toString(profit));
34 }
```

Fig. 1. A motivating example

Execution in the caller continues in parallel with the new process. The value of a method call is a *future* (see Line 30), which can be used to synchronize with the resulting process (Line 31) and to obtain the result (Line 32). Abbreviated syntax makes it possible to omit an explicit future definition to synchronize with the callee and, optionally, obtaining the result (see Line 18).

One question is what the behavior of an asynchronous method call is, when it immediately followed by a `fp.get` expression, e.g., omitting Line 31 in the example. In this case, the `get` expression *blocks* until `fp` has a value. Blocking means that the cog will not schedule another process. There exists abbreviated syntax for this kind of call: instead of `f = o!m(); v = f.get;` one can write `v = o.m();`. This notation is used in examples later in this paper.

Finally, the behavior of a model is specified via its *main block* (Lines 24–34).

2.2 Asynchronous Method Calls, Scheduling Points, and Object Groups

The concurrency model of ABS merits some more explanation. The unit of concurrency in ABS is the *concurrent object group* (cog). Each cog contains a number of objects and cooperatively schedules the processes running on these objects

such that at most one process per cog is running. As mentioned in Sect. 2.1, each asynchronous method call results in a process being created at callee-side that executes the method named in the call. Figure 2 shows the relation of processes, cogs, and sessions (sessions are introduced in Sect. 3.4). So, the two processes created by the method calls in Fig. 1, Line 20 and 21 can run in parallel provided they are not running in the same cog.

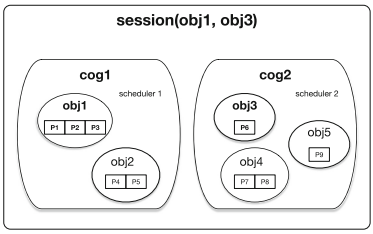


Fig. 2. Cogs contain objects, which run processes. A session temporarily "captures" its set of participants. The session names participating objects whose cogs join the session; other objects in the cog cannot join a different session at the same time.

A cog schedules a process to run when its currently running process reaches a *scheduling point*. A scheduling point occurs when a process terminates, either by executing its return statement or via an unhandled exception, or at the point of an `await` or `suspend` statement. The cog will choose the next process to run non-deterministically from its set of *runnable processes*. A process is runnable after it has been freshly created, after a `suspend` statement, and after an `await` statement if the condition in the `await` statement is true.

Cogs and cooperative scheduling makes modeling distributed concurrent systems easy and safe. Processes in different cogs are running in parallel, but do not have access to shared state. Processes within the same cog, on the other hand, can share state if they run on the same object, but are running interleaved, with scheduling points clearly visible at the source code level.

3 Exception Recovery in ABS

This section describes the new constructs added to the ABS language for modeling exceptional situations, handling exceptions and recovering from unhandled exceptions, and multi-party sessions.

The current ABS language documentation can be found at [1]. A formal semantics of ABS can be found in [13]. Figures 3 and 4 summarize the syntax of the ABS functional and imperative layer, respectively. Parts highlighted in yellow mark the elements added in this paper.

Syntactic categories	*Definitions*
T in Ground Type	$T ::= B \mid I \mid D \mid D\langle\overline{T}\rangle \mid E$
B in Basic Type	$B ::= \texttt{Bool} \mid \texttt{Int} \mid \cdots$
A in Type	$A ::= N \mid T \mid D\langle\overline{A}\rangle$
N in Name	$Dd ::= \texttt{data}\ D[\langle\overline{A}\rangle] = Cons[\overline{\mid Cons}];$
E in Exception	$Cons ::= Co[(\overline{A})]$
x in Variable	$E ::= \texttt{exception}\ Co[(\overline{A})];$
e in Expression	$F ::= \texttt{def}\ A\ fn[\langle\overline{A}\rangle](\overline{A}\ \overline{x}) = e;$
b in Bool Expression	$e ::= b \mid x \mid t \mid \texttt{this} \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \texttt{case}\ e\ \{\overline{br}\}$
t in Ground Term	$t ::= Co[(\overline{t})] \mid \texttt{null}$
br in Branch	$br ::= p \Rightarrow e;$
p in Pattern	$p ::= \_ \mid x \mid t \mid Co[(\overline{p})]$

Fig. 3. Core ABS syntax for the functional level. Terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over corresponding syntactic categories, square brackets [] denote optional elements. (Color figure online)

Syntactic categories.	*Definitions.*
s in Stmt	$P ::= \overline{IF}\ \overline{CL}\ \{[\overline{T}\ \overline{x};]\ s\}$
e in Expr	$IF ::= \texttt{interface}\ I\ \{\ [\overline{Sg}]\ \}$
b in BoolExpr	$CL ::= \texttt{class}\ C\ [(\overline{T}\ \overline{x})]\ [\texttt{implements}\ \overline{I}]\ [\texttt{recover}\ \{\ \overline{cbr}\ \}]\ \{\ [\overline{T}\ \overline{x};]\ \overline{M}\}$
g in Guard	$Sg ::= T\ m\ ([\overline{T}\ \overline{x}])$
cbr in Catch branch	$M ::= Sg\ \{[\overline{T}\ \overline{x};]\ s\}$
	$s ::= \{\ s\ \} \mid s; s \mid \texttt{skip} \mid x = rhs \mid \texttt{if}\ b\ \{\ s\ \}\ [\texttt{else}\ \{\ s\ \}] \mid \texttt{while}\ b\ \{\ s\ \}$
	$\mid \texttt{suspend} \mid \texttt{await}\ g \mid \texttt{return}\ e$
	$\mid \texttt{try}\ s\ \texttt{catch}\ \{\ \overline{cbr}\ \}\ [\texttt{finally}\ s\ ] \mid \texttt{throw}\ e$
	$\mid \texttt{session}(\overline{e})\{\ s\ \}\ [\texttt{recover}\ \{\ \overline{cbr}\ \}\ ]$
	$rhs ::= e \mid e.m(\overline{e}) \mid e!m(\overline{e}) \mid x.\texttt{get} \mid \texttt{new}\ [\texttt{local}]\ C\ (\overline{e})$
	$cbr ::= p\ \texttt{=>}\ s$
	$g ::= b \mid x?$

Fig. 4. Syntax for the imperative layer of ABS. Notation as in Fig. 3 (Color figure online)

3.1 Exception Modeling in the Functional Layer

Algebraic data structures in ABS are defined with the keyword `data`, which defines both a type and a set of constructors. Exceptions are defined with the keyword `exception`, which introduces a named constructor for the new exception. The type of an exception is always `ABS.StdLib.Exception`, which is predefined in the ABS standard library. Exceptions can be used as data values. For example, they can be stored in lists and can be used in the `case` pattern-matching expression. Additionally, exceptions are used as argument to the `throw` statement and are pattern-matched in `catch` branches (see Sect. 3.2 below).

3.2 Exception Handling in the Imperative Layer

The imperative layer of ABS adds a `throw` statement for manually raising exceptions. Additionally, normal code execution can also lead to exceptions, like attempting to send a message to `null` or dividing by zero.

For handling exceptions, the imperative layer of ABS adds the familiar `try-catch-finally` construct. Exceptions raised in the statement(s) protected by `try` are pattern-matched by the branches in the `catch` block; the statements in the first matching branch are then executed ("the exception is *handled* by that branch"). Finally, all statements in the `finally` block are executed, regardless of how the `try` block was executed. In case no catch branch matches ("the exception is *unhandled*"), the `finally` block is executed and the exception is (hopefully) handled by an enclosing `try-catch` block. The scope of variables declared in the `try` block does not extend to the `catch` and `finally` blocks since they might not have been initialized yet when entering these blocks. To ensure progress, `finally` blocks cannot contain blocking operations or process suspension.

Unhandled exceptions terminate the current process and are stored in its future. As in [10], unhandled exceptions propagate across futures. When the callee process terminated with an exception, that exception will be raised when the caller tries to obtain the future's value via a `get` expression, and will thereby propagate along the chain of process invocations until it is handled.

Note that a process crash is effectively ignored if no other process tries to access its return value.

3.3 Recovery in the Object Layer

The compositional proof system of ABS [5,7] relies on class invariants; processes are responsible to establish these invariants at all of their scheduling points. Since with exceptions processes can terminate at arbitrary points, we introduce *recovery blocks* as a fall-back mechanism to reestablish class invariants.

All unhandled exceptions still lead to process termination, as above in Sect. 3.2, but additionally the unhandled exception is matched against the recovery block given in the class definition. If a matching branch is found, its statements are executed and the object is kept alive. If no matching branch is found in the recovery block, the object is killed. A dead object is marked as invalid, all processes running on it are terminated, and all further messages to that object result in an exception in the caller. This is not quite as draconian as it sounds, since models of distributed systems need to model this type of partial failure anyway.

3.4 Session Blocks

As discussed above, try-catch blocks and class recovery blocks help restore per-object class invariants in the face of exceptional situations. But they do not help in a systematic way for recovering invariants that span more than one object. In general, this requires corrective actions undoing or compensating from messages sent as part of an incomplete transaction. For example, see Fig. 1: when creating a `CTransaction` object with `factor`= 0, executing the `transfer` method will lead to a division by zero on Line 19, after sending a `withdraw` message to `sender` but before the corresponding `deposit` messages. Hence, the system-wide invariant

("the amount of money in the system is constant") is violated. To handle these cases, we introduce the `session block` construct.

A *session* is the analogue of a critical section over a group of cogs. During the lifetime of a session, the participating cogs will only run processes that "belong" to the session. Unrelated processes are not scheduled until the session has ended. Sessions are implemented and modeled via *session blocks*. The cog running the process that is executing the session block (the "session initiator") is a *session participant*, as are the cogs of all objects named in the session block parameter list. In Fig. 2 we see a session with two participants. For the duration of a session, all participants will only schedule processes that are created during the session by a session participant. There can be multiple active sessions in the system, but no cog can participate in more than one session at a time.

Figure 4 introduces the syntax of the `session` block. Figure 5 shows an example of this construct, in a revised `CTransaction` class. Note the use of local variables `start_sender` etc. is to record progress through the session and establish which actions in the `CAccount` objects can be undone. As with `try-catch-finally`, variables declared in the body of the session block go out of scope before entering the recovery block, since their value and status are uncertain.

The semantics of initiating and terminating a session demands synchronization among all participants. When the session initiator starts executing a session block, the list of participating cogs is calculated from the block's parameter list.

```
1  class CTransaction(Account commission, Rat factor) implements Transaction {
2    Unit transfer (Account sender, Account receiver, Rat amount) {
3      Maybe<Rat> start_sender = Nothing;
4      Maybe<Rat> start_receiver = Nothing;
5      Maybe<Rat> start_commission = Nothing;
6      session(sender, receiver, commission) {
7        Rat sb = sender.getBalance(); start_sender = Just(sb);
8        Rat rb = receiver.getBalance(); start_receiver = Just(rb);
9        Rat cb = commission.getBalance(); start_commission = Just(cb);
10       sender.withdraw(amount);
11       Rat profit = amount / factor;
12       commission.deposit(profit);
13       receiver.deposit(amount - profit);
14     } recover {
15       _ => {
16         if (isJust(start_sender)) {
17           Rat bal_sender = sender.getBalance();
18           sender.deposit(fromJust(start_sender) - bal_sender);
19         }
20         if (isJust(start_commission)) {
21           Rat bal_commission = commission.getBalance();
22           commission.withdraw(bal_commission - fromJust(start_commission));
23         }
24         if (isJust(start_receiver)) {
25           Rat bal_receiver = receiver.getBalance();
26           receiver.withdraw(bal_receiver - fromJust(start_receiver));
27         }
28       }
29     }
30   }
31 }
```

Fig. 5. Error recovery in the transaction class via a session block

In Fig. 5, Line 6, there are four participants (the cogs of the three `Account` objects plus the cog of the `Transaction` object, which runs the session initiator). Execution of the initiating process blocks until all participants have (a) left any currently active sessions they might be in, and (b) have reached a scheduling point. Then, all participants acknowledge entering the session and receive the list of participants. When the session initiator reaches the end of the session body, either normally or via an exception, execution blocks until all participants have finished executing all processes that are part of the session. A final synchronization point is at the end of the recovery block, in case it is entered.

4 Program Analysis of Session Blocks with Exception Handlers

A session block, introduced in Sect. 3, is used to identify a special group of interactions in which (i) the states of participants in the interactions shall not be updated by other processes, and (ii) once an exception occurs but is not caught by `catch` block, the recovery block will recover the states of participants.

In this section, we give a session-contract based verification framework. This verification framework is inspired by the ABS class-invariant based verification [5], which, however, is not designed for verifying the preservation of invariants while exceptions are thrown or verifying properties across multiple objects, such as in the case in Fig. 1.

4.1 Session Contracts

In this section we first briefly explain the class-invariant based verification framework for ABS [7]. Then we will point out why this proof strategy is too strong for the language setting where exception handling is considered. The verification framework in [7] assumes formal specification at the class level, i.e. for each object implemented in a class C we aim to establish its class invariant I_{C}. We need to prove that C's initialization block establishes I_{C}, and I_{C} holds before process releasing at each `await` and `suspend` statements, as well as when a method on C returns. Thus, class invariants need to hold at each scheduling points but not necessary in between. Consequently, if an exception is thrown between two scheduling points, this may lead to a system ending in a state where class invariant does not hold. For instance, we define a specification for the banking example in Fig. 1.

$$\texttt{sender.balance} + \texttt{commission.balance} + \texttt{receiver.balance} = v \qquad (1)$$

which says that the summation of balances of the sender's account, the receiver's account and the commission's account is a constant v. This property cannot be proven within the verification framework for ABS [7]. One reason is that the specification language used in [7] cannot express the state of the invoked objects, and this property does not hold at every suspension point as it should in [7], for

instance, after the balance of the `sender` has been decreased but the balance of other accounts have not yet been changed. Besides, if there is any runtime error, for example division by zero, this property does not hold when an exception is thrown.

To overcome these restrictions, the concept of *session* is introduced in this work. The modified version of the banking example using *session* is presented in Fig. 5, for which we define Eq. (1) as a *session contract*. Session contracts express the state of the session or the communication pattern between objects in the same session. They are assumed at session entry and should be proven at session exit. Accordingly, the following statement should be proven upon session termination in Fig. 5.

$$v - \texttt{amount} + \texttt{profit} + (\texttt{amount} - \texttt{profit}) = v$$

In case of uncaught exceptions in the session block, the session contract should hold after the recovery block. In order to prevent the session state from being randomly modified at the process release points, we only allow process suspension outside the session blocks.

4.2 Proof System

In this section we introduce a modular proof system for proving session-based ABS programs. We first prove that each method satisfies its method contract and then prove that each session block satisfies its corresponding session contract.

4.2.1 Program Analysis at Method Level

We verify ABS methods against method contracts by applying the proof rules in Fig. 6, i.e. one rule for each program statement. The program logic is first-order dynamic logic for ABS (ABSDL) [2,5,7]. For a sequence of executable ABS statements $\mathcal{S}$ and ABSDL formulae P and Q, the formula $P \Rightarrow [\mathcal{S}]Q$ expresses: If the execution of $\mathcal{S}$ starts in a state where the assertion P holds and the program terminates normally, then the assertion Q holds in the final state. Gentzen-style sequent calculus is used to prove ABSDL formulae. In sequent notation, $P \Rightarrow [\mathcal{S}]Q$ is written $P \vdash [\mathcal{S}]Q$. A sequent calculus as realized in ABSDL essentially constitutes a symbolic interpreter for ABS. For example, the method rule in Fig. 6 expresses the proof of method `m` against its precondition p and post-condition q. In the assign rule, the assignment $v = e$ is an active statement in a modality $[\pi\ v = e; \omega]$, where v is a program variable and e is a pure (side effect-free) expression. The nonactive prefix π consists of an arbitrary sequence of opening braces "{", i.e. beginnings "$\texttt{m}(\overline{\texttt{x}})\{$" of method blocks, "`try`{" of `try-catch-finally` blocks, and "$\texttt{session}(\overline{e})\{$" of session blocks. The remaining program is represented by ω. The assign rule generates a so-called update [2], as $\{v := e\}$ shown above, for the assignment statement, which captures state changes and is placed outside the modality box. Updates can be viewed as explicit substitutions that accumulate in front of the modality during symbolic program execution. We use $\mathcal{U}$ to represent the accumulated updates up to now. Updates can only be

applied to formulae or terms. Once the program to be verified has been completely executed and the modality is empty (see the emptyBox rule in Fig. 6), the accumulated updates are applied to the formula after the modality, resulting in a pure first-order formula. Γ stands for (possibly empty) sets of side formulae, and ϕ the property required to be proven upon execution termination.

$$\textsf{assign}\ \frac{\Gamma \vdash \mathcal{U}\{v := e\}[\pi\ \omega]\phi}{\Gamma \vdash \mathcal{U}[\pi\ v = e;\omega]\phi} \qquad \textsf{skip}\ \frac{\Gamma \vdash \mathcal{U}[\pi\ \omega]\phi}{\Gamma \vdash \mathcal{U}[\pi\ \texttt{skip};\omega]\phi}$$

$$\textsf{new}\ \frac{\Gamma, cl(o,C) \vdash \mathcal{U}(fresh(o) \Rightarrow \{v := o\}[\pi\ \omega]\phi)}{\Gamma \vdash \mathcal{U}[\pi\ v = \texttt{new}\ C(\overline{e});\omega]\phi} \qquad \textsf{method}\ p \vdash [\mathsf{m}(\overline{\mathsf{x}})\{s\}]q$$

$$\textsf{ifElse}\ \frac{\begin{array}{c}\Gamma \vdash \mathcal{U}(b \Rightarrow [\pi\ s_1\ \omega]\phi)\\ \Gamma \vdash \mathcal{U}(\neg b \Rightarrow [\pi\ s_2\ \omega]\phi)\end{array}}{\Gamma \vdash \mathcal{U}[\pi\ \texttt{if}\ b\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}\ \omega]\phi} \qquad \textsf{while}\ \frac{\begin{array}{c}\Gamma \vdash \mathcal{U}I\\ \Gamma, I \wedge b \vdash [s]I\\ \Gamma, I \wedge \neg b \vdash [\pi\ \omega]\phi\end{array}}{\Gamma \vdash \mathcal{U}[\pi\ \texttt{while}\ b\ \{\ s\ \};\omega]\phi}$$

$$\textsf{return}\ \frac{\Gamma, C.\mathsf{m}(\overline{\mathsf{x}}) : (p,q) \vdash (\mathcal{U}\{\mathtt{r} := e\}q) \wedge \mathcal{U}[\pi\ \omega]\phi}{\Gamma, C.\mathsf{m}(\overline{\mathsf{x}}) : (p,q) \vdash \mathcal{U}[\pi\ \texttt{return}\ e;\omega]\phi} \qquad \textsf{emptyBox}\ \frac{\Gamma \vdash \mathcal{U}\phi}{\Gamma \vdash \mathcal{U}[\,]\phi}$$

$$\textsf{asyncCall}\ \frac{\begin{array}{c}\Gamma, cl(o,C), C.\mathtt{m}(\overline{\mathsf{x}}) : (p,q) \vdash \mathcal{U}\{\mathsf{this} := o\}\{\overline{\mathsf{x}} := \overline{e}\}p\\ \Gamma, cl(o,C), C.\mathtt{m}(\overline{\mathsf{x}}) : (p,q), bt(fr,o,C.\mathtt{m}(\overline{e})) \vdash \mathcal{U}\{fr := fr'\}[\omega]\phi\end{array}}{\Gamma, cl(o,C), C.\mathtt{m}(\overline{\mathsf{x}}) : (p,q) \vdash \mathcal{U}[fr = o!\mathtt{m}(\overline{e});\omega]\phi}$$

$$\textsf{get}\ \frac{\begin{array}{c}\Gamma, bt(fr,o,C.\mathtt{m}(\overline{e})), C.\mathtt{m}(\overline{\mathsf{x}}) : (p,q), isValue(fr), fresh(v') \vdash\\ \qquad (\mathcal{U}\{\mathsf{this} := o\}\{\overline{\mathsf{x}} := \overline{e}\}\{\mathtt{r} := v'\}q) \Rightarrow \mathcal{U}\{v := v'\}[\pi\ \omega]\phi\\ \Gamma, bt(fr,o,C.\mathtt{m}(\overline{e})), C.\mathtt{m}(\overline{\mathsf{x}}) : (p,q), \neg isValue(fr), fresh(v') \vdash \mathcal{U}\{v := v'\}[\pi\ \texttt{throw}\ v;\ \omega]\phi\end{array}}{\Gamma, bt(fr, C.\mathtt{m}(\overline{e})), C.\mathtt{m}(\overline{\mathsf{x}}) : (p,q) \vdash \mathcal{U}[\pi\ v = fr.\mathtt{get};\omega]\phi}$$

$$\textsf{syncCall}\ \frac{\Gamma, \mathcal{U}fresh(fr') \vdash \mathcal{U}[\pi\ fr' = o!\mathtt{m}(\overline{e}); v = fr'.\mathtt{get};\omega]\phi}{\Gamma \vdash \mathcal{U}[\pi\ v = o.\mathtt{m}(\overline{e});\omega]\phi}$$

Fig. 6. Proof rules for statements.

Figure 6 also provides rules for other statements. Rules skip, new, return, and get are for `skip` statements, object creation, return statements, and `get` statements, respectively. In rule `new`, $fresh(o)$ expresses that there is no object reference equals o up to now. Object o belongs to class C is captured by predicate $cl(o, C)$. The rule ifElse is for conditional statements. The rule while proves that a while loop preserves loop invariant I. In the rule asyncCall we assume that method contract of the invoked method is provided. We formulate method contract in the form of $C.\mathsf{m}(\overline{\mathsf{x}}) : (p, q)$, where (p, q) is a pair of pre- and post-condition of method $\mathsf{m}(\overline{\mathsf{x}})$ in class C. For brevity, we skip the case of multiple implementations of a given interface but they can be handled in the standard way using adaptation rule [6]. The asyncCall rule has two premises. The first one proves that the precondition p of m holds. The update substitutes this with callee o, and formal parameters $\overline{\mathsf{x}}$ with actual parameters $\overline{e}$. In the second premise, a fresh future fr' is generated and added into an update clause. The environment carries information about the callee of fr', i.e. predicate $bt(fr, o, C.\mathtt{m}(\overline{e}))$ expresses that future fr belongs to method $\mathtt{m}(\overline{e})$ which is executed on object o of class C. In rule return, the keyword r captures the return value and the postcondition q is required to be proven. Note that we consider partial correctness, so for the

get rule we assume it is possible to fetch the data from the future at the `get` statements eventually. Since it is not possible to know the exact fetched data while applying the get rule, we follow the same principle as for the new rule and assign a fresh value, i.e. v', to variable v. However, if the environment carries information about the callee of fr, we can use the post condition q to restrict the possible values v'. If such information is unavailable, we assume $q = true$. If future fr in the `get` statement does not contain value, i.e. $\neg isValue(fr)$, but an exception, an exception will be thrown. This is captured by the second premise. The rule syncCall is syntactic sugar to an asynchronous call plus a `get` statement. Note that we do not present the proof rules for `await` and `suspend` statements in this paper, because we only allow process suspension outside the session blocks.

$$\text{try-catch-finally}\ \frac{\begin{array}{c}\Gamma, e = t \vdash \mathcal{U}[\pi\ \texttt{try}\{\overline{s_2}\}\ \texttt{catch}\{\}\ \texttt{finally}\{\overline{s_3}\}\ \omega]\phi \\ \Gamma, e \neq t \vdash \mathcal{U}[\pi\ \texttt{try}\{\texttt{throw}\ e;\ \overline{s_1}\}\ \texttt{catch}\{\overline{cbr}\}\ \texttt{finally}\{\overline{s_3}\}\ \omega]\phi\end{array}}{\Gamma \vdash \mathcal{U}[\pi\ \texttt{try}\{\texttt{throw}\ e;\ s_1\}\ \texttt{catch}\{t \Rightarrow \overline{s_2}; \overline{cbr}\}\ \texttt{finally}\{\overline{s_3}\}\ \omega]\phi}$$

$$\text{try-emptyCatch-finally}\ \frac{\Gamma \vdash \mathcal{U}[\pi\ \overline{s_2}; \texttt{throw}\ e\ \omega]}{\Gamma \vdash \mathcal{U}[\pi\ \texttt{try}\{\texttt{throw}\ e;\ \overline{s_1}\}\ \texttt{catch}\{\}\ \texttt{finally}\{\overline{s_2}\}\ \omega]\phi}$$

$$\text{emptyTry}\ \frac{\Gamma \vdash \mathcal{U}[\pi\ \overline{s}\ \omega]\phi}{\Gamma \vdash \mathcal{U}[\pi\ \texttt{try}\{\}\ \texttt{catch}\{\overline{cbr}\}\ \texttt{finally}\{\overline{s}\}\ \omega]\phi}$$

Fig. 7. Proof rules for `try-catch-finally` statements.

In Fig. 7 we provide proof rules for try-catch-finally statements. Runtime exceptions are handled in the proof rules (see the example of the get rule in Fig. 6). Errors created during evaluation of expressions, for example division by zero, are handled in a similar way. The rule try-catch-finally has two premises. If the thrown exception matches the first case listed in the catch block, statements s_2 from the catch clause are then executed. Otherwise, the exception is thrown again and the first case in the catch block is eliminated. Note that the `finally` block can be empty, i.e. s_3 is an empty list. The rule try-emptyCatch-finally expresses that the exception cannot be caught by the `catch` clause so it executes the `finally` clause and then throws the exception to the outer scope. Maybe there will be another `try-catch` clause around it, i.e. contained in the nonactive code π and the remaining program ω. The rule emptyTry says if a `try` clause is completely executed without throwing any exceptions, then the `finally` clause and the remaining program will be executed.

4.2.2 Proof of the Example at Method Level

In this section, we provide method contracts for the example shown in Sect. 3.4. The method contract for `withdraw` is

$$\texttt{CAccount.withdraw(Rat amount}_1\texttt{)}:(\texttt{this.balance} = \texttt{balance}', \texttt{this.balance} = \texttt{balance}' - \texttt{amount}_1)$$

in which this.balance accesses the field `balance`, and logical variable $\texttt{balance}'$ stores the value of `balance` at the prestate. This contract expresses that if the

value of `balance` at prestate is $\texttt{balance}'$, then it updates to $\texttt{balance}' - \texttt{amount}_1$ upon method termination. The method contract for `deposit` is:

$$\texttt{CAccount.deposit(Rat amount}_2\texttt{):(this.balance} = \texttt{balance}'', \texttt{ this.balance} = \texttt{balance}'' + \texttt{amount}_2\texttt{)}$$

This contract expresses that if the value of `balance` at prestate is $\texttt{balance}''$, then it updates to $\texttt{balance}'' + \texttt{amount}_2$ upon method termination. Finally, a method contract for `getBalance` is given below:

$$\texttt{CAccount.getBalance():(true, r} = \texttt{this.balance)}$$

There is no requirement for the precondition. In the postcondition the value of `balance` is assigned to variable `r`. We can prove these three method contracts by using rules method, assign, return and emptyBox in Fig. 6.

4.2.3 Program Analysis at Session Level

Session contract must be proven at the end of the session if there is no exception left unhandled, assuming the session contract holds at the session entry. The proof rules in Fig. 8 together with the ones in Figs. 6 and 7 build up a proof system for verifying session blocks. The rules in Figs. 6 and 7 will be used when the session block is not empty and the current active statement is not a `throw` statement for exception. A session block may contain `try-catch-finally` clauses but cannot be nested within another session block.

$$\textsf{sessionRecover}\ \frac{\Gamma, e = t \vdash \mathcal{U}[\texttt{session}(\overline{e})\{\overline{s_2}\}\ \texttt{recover}\{\}]\phi \quad \Gamma, e \neq t \vdash \mathcal{U}[\texttt{session}(\overline{e})\{\texttt{throw}\ e;\ \overline{s_1}\}\ \texttt{recover}\{\overline{cbr}\}]\phi}{\Gamma \vdash \mathcal{U}[\texttt{session}(\overline{e})\{\texttt{throw}\ e;\ \overline{s_1}\}\ \texttt{recover}\{t \Rightarrow \overline{s_2}; \overline{cbr}\}]\phi}$$

$$\textsf{emptyRecover}\ \frac{\Gamma \vdash \texttt{false}}{\Gamma \vdash \mathcal{U}[\texttt{session}(\overline{e})\{\texttt{throw}\ e;\ r\}\ \texttt{recover}\{\}]\phi}$$

$$\textsf{emptySession}\ \frac{\Gamma \vdash \mathcal{U}\phi}{\Gamma \vdash \mathcal{U}[\texttt{session}(\overline{e})\{\}\ \texttt{recover}\{\overline{s}\}]\phi}$$

$$\textsf{sessionStart}\ SC, \overline{C.\mathsf{m}(\overline{\mathsf{x}}) : (p, q)}, \mathit{init} \vdash [\texttt{session}(\overline{e})\{\overline{s_1}\}\ \texttt{recover}\{\overline{s_2}\}]SC$$

Fig. 8. Rules for proving session contracts.

The rule sessionRecover has two premises. If the thrown exception matches the first case listed in the `recover` block, statements s_2 from the `recover` clause are then executed. Otherwise, the same exception is thrown again and the first case in the `recover` block is eliminated. The rule emptyRecover says if none of the cases in the `recover` block matches the exception thrown from a `session` block, this proof branch cannot be closed. This means the `recover` block needs to be re-implemented until the session contract can be successfully proven in the end of the `recover` block. The rule emptySession says if a session block is completely executed without throwing any exceptions, then the session contract should be proven at this session exit. The rule sessionStart captures the proof obligation of a session. We use this rule to prove that a session preserves the

session contract *SC*. Contracts for all the methods in the system are assumed known from the beginning. The set *init* are variables defined before session entry but used in the session block.

4.2.4 Proof of the Example at Session Level

In this section, we explain the proof outline for the example in Fig. 5, which presents the cases when exceptions are thrown in a session. Equation (1) is the corresponding session contract. Since the session involves method invocations, the proof requires knowledge of all the invoked methods from the session. This knowledge is formalized as the method contracts presented in Sect. 4.2.2.

An exception can be thrown at any execution point of the session block. Since there is an execution barrier between the session body and the recovery block (see Sect. 3.4), all the processes executed in or related to the session block are finished before the recovery block can be executed. The recovery block rescues all the possible failing cases and makes sure the program is back to the state as if this particular transaction has never been executed. Below we present the proof outline for the recovery block and show that the session contract holds at the end of the recovery block.

Session participants, i.e. objects, and their method invocations are known in a session. According to this knowledge, we instantiate each method contract of the invoked methods in the session as follows: In the method contract for the `deposit` method of the `sender` object, we instantiate this to `sender` and parameter $\texttt{amount}_2$ to start_sender − bal_sender. In the method contract for the `withdraw` method of the `commission` object, we instantiate this to `commission` and parameter $\texttt{amount}_1$ to bal_commission − start_commission; In the method contract for the `withdraw` method of the `receiver` object, we instantiate this to `receiver` and parameter $\texttt{amount}_1$ to bal_receiver − start_receiver.

```
sender.deposit(start_sender − bal_sender):
  (sender.balance = bal_sender,
    sender.balance = bal_sender + (start_sender − bal_sender))

commission.withdraw(bal_commission − start_commission):
  (commission.balance = bal_commission,
    commission.balance = bal_commission − (bal_commission − start_commission))

receiver.withdraw(bal_receiver − start_receiver):
  (commission.balance = bal_receiver,
    commission.balance = bal_receiver − (bal_receiver − start_receiver))
```

Assume in the beginning of the session block the following holds

$$\texttt{sender.balance} = sb \wedge \texttt{commission.balance} = cb \wedge \texttt{receiver.balance} = rb$$

where sb, cb, rb are logical variables to record the initial balance of the accounts and $sb + cb + rb = v$. Besides, start_sender $= sb$ when established, start_commission $= cb$ when established, and start_receiver $= rb$ when established. Depending on which initial balance of the accounts are successfully accessed in the session block, the conditional branches in the recovery block will be selected

for execution. In the end of the recovery block we show that

$$\begin{aligned}&\texttt{sender.balance} + \texttt{commission.balance} + \texttt{receiver.balance}\\ &= [sb \mid \mathsf{bal\_sender} + (\mathsf{start\_sender} - \mathsf{bal\_sender})] +\\ &\quad [cb \mid \mathsf{bal\_commission} - (\mathsf{bal\_commission} - \mathsf{start\_commission})] +\\ &\quad [rb \mid \mathsf{bal\_receiver} - (\mathsf{bal\_receiver} - \mathsf{start\_receiver})]\\ &= sb + cb + rb = v\end{aligned}$$

in which the symbol $[a \mid b]$ means either a is selected or b is selected. Thus, this session contract does hold in the end of the recovery block. From these proof results, we show that the execution of a session will always end in a safe state, with the session contract reestablished, irregardless of the presence of exceptions.

5 Related Work

The use of futures for transferring both result and exception values goes back to [15]. A specification of an exception handling system for active objects using one-way asynchronous communication and interacting via a request/response protocol was presented in [9]. Future-based communication and verification of such system were not considered. The first approach for exception recovery based on object state rollback for unhandled errors in ABS was proposed in [10]. This approach was implemented but ultimately rejected for inclusion in the main language for two reasons: the impact of object rollback on the ABS proof theory was too high, and the proposed approach did not handle notions of correctness that need to be expressed over multiple objects. The approach in this paper addresses both of these shortcomings. Reasoning about exception handling in Java is supported by PVS [12]. The invariant-based verification framework for ABS was provided by [5,7], in which formal specification was at the class level. KeY-ABS [5] is a theorem prover that realized the proof system for ABS. It was developed based on KeY [2], which supports deductive verification for exception handling in sequential Java programs. A new rule implemented in KeY to reason about exception handling in loops was introduced by [16]. A modular and scalable network-on-chip example is proven by KeY-ABS. The proof results are shown in [8]. In the work of [14], core ABS is extended with sessions and annotations to express scheduling policies based on required communication ordering. The annotation is statically checked against the session automata derived from the session types.

6 Conclusion

This paper shows an extension of the Active Object-based modeling language ABS with exception specifications, handling and recovery. We introduce several language constructs including the means to express coordinated multi-party sessions and recovery actions. To guarantee session correctness, we provide session

contracts and an attendant proof system. We show that a system, in which exceptions are thrown, can be recovered back to a safe state, where session contract holds. Soundness proofs for the reasoning system with respect to the operational semantics are left as future work. Other planned future work includes (1) building up our type system to describe allowed scheduling during the lifetime of the session, and (2) tool support for the new language constructs, including an implementation of the type checking and runtime semantics, and a thorough evaluation of the usability of session contracts in the context of existing case studies utilizing ABS. Inspired by behavioral types [3,11], our type system will be designed to regulate the runtime behavior of objects and schedulers, and reduce the number of exceptions caused by undesired communication behaviour.

Acknowledgement. This work was supported by the research projects CUMULUS: Semantics-based Analysis for Cloud-Aware Computing, ERC project LiveSoft, and the SIRIUS Centre for Scalable Data Access.

References

1. The ABS Development Team. ABS Documentation. http://docs.abs-models.org
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
3. Chen, T.-C., Viering, M., Bejleri, A., Ziarek, L., Eugster, P.: A type theory for robust failure handling in distributed systems. In: Albert, E., Lanese, I. (eds.) FORTE 2016. LNCS, vol. 9688, pp. 96–113. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39570-8_7
4. de Boer, F.S., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., Yang, A.M.: A survey of active object languages. ACM Comput. Surv. **50**(5), 76:1–76:39 (2017)
5. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: a deductive verification tool for the concurrent modelling language ABS. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 517–526. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_35
6. Din, C.C., Johnsen, E.B., Owe, O., Yu, I.C.: A modular reasoning system using uninterpreted predicates for code reuse. J. Log. Algebraic Methods Program. **95**, 82–102 (2018)
7. Din, C.C., Owe, O.: Compositional reasoning about active objects with shared futures. Formal Aspects Comput. **27**(3), 551–572 (2015)
8. Din, C.C., Tapia Tarifa, S.L., Hähnle, R., Johnsen, E.B.: History-based specification and verification of scalable concurrent and distributed systems. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 217–233. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25423-4_14
9. Dony, C., Urtado, C., Vauttier, S.: Exception handling and asynchronous active objects: issues and proposal. In: Dony, C., Knudsen, J.L., Romanovsky, A., Tripathi, A. (eds.) Advanced Topics in Exception Handling Techniques. LNCS, vol. 4119, pp. 81–100. Springer, Heidelberg (2006). https://doi.org/10.1007/11818502_5

10. Göri, G., Johnsen, E.B., Schlatte, R., Stolz, V.: Erlang-style error recovery for concurrent objects with cooperative scheduling. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. LNCS, vol. 8803, pp. 5–21. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45231-8_2
11. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 31–336 (2016)
12. Jacobs, B.: A formalisation of Java's exception mechanism. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, pp. 284–301. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45309-1_19
13. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
14. Kamburjan, E., Din, C.C., Chen, T.-C.: Session-based compositional analysis for actor-based languages using futures. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM 2016. LNCS, vol. 10009, pp. 296–312. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47846-3_19
15. Liskov, B., Shrira, L.: Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI), pp. 260–267 (1988)
16. Steinhöfel, D., Wasser, N.: A new invariant rule for the analysis of loops with non-standard control flows. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 279–294. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_18
17. Waldo, J., Wyant, G., Wollrath, A., Kendall, S.: A note on distributed computing. In: Vitek, J., Tschudin, C. (eds.) MOS 1996. LNCS, vol. 1222, pp. 49–64. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-62852-5_6

Spread the Work: Multi-threaded Safety Analysis for Hybrid Systems

Stefan Schupp and Erika Ábrahám(✉)

Theory of Hybrid Systems, RWTH Aachen University, Aachen, Germany
{stefan.schupp,abraham}@cs.rwth-aachen.de

Abstract. We consider a method for the bounded safety analysis of hybrid systems, whose continuous behaviour is intertwined with discrete execution steps. The method computes a tree of state sets, which together over-approximate reachability by bounded-length executions. If none of the state sets intersects with a given set of unsafe states then we have proven bounded safety. Otherwise, we iteratively repeat parts of the computations with locally refined search parameters, in order to reduce the over-approximation error.

In this paper we present a parallelization technique for the above method. We identify independent computations that can be carried out by different threads/processes concurrently, and examine how to achieve work-balance between the threads at low communication cost. Furthermore, we discuss how to assure mutually exclusive node access during refinement computations, without high synchronization costs. We evaluate our proposed solutions experimentally on some benchmarks.

1 Introduction

The massive application of digital controllers for the control of continuous (e.g. physical) systems raises the need for verification approaches for such *hybrid* systems with mixed discrete-continuous behaviour. Though the reachability problem for hybrid systems is in general undecidable, a variety of incomplete safety analysis approaches have been developed. Besides verification methods based on theorem proving, SMT solving or rigorous simulation, these include techniques based on *flowpipe construction*, e.g. [1–3].

Starting from a set of initial states, flowpipe-construction-based methods iteratively over-approximate *flowpipes*, i.e. the set of states reachable from a given state set via time evolution according to the system's continuous dynamics, and sets of successors via discrete execution steps. Due to non-determinism, these computations generate a tree with state sets as nodes, where the root includes all initial states and the children of a node include all discrete successors from the node's flowpipe. These computations are usually bounded in the time duration

This work was supported by the German research council (DFG) in the context of the HyPro project and the DFG Research Training Group 2236 UnRAVeL.

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 89–104, 2018.
https://doi.org/10.1007/978-3-319-92970-5_6

for flowpipes and the number of discrete steps executed (unless a fixedpoint can be detected).

If none of the flowpipes and discrete successor sets contain unsafe states then the model is safe. Otherwise, due to over-approximation, no conclusive information can be derived. Therefore, it is important to provide possibilities to reduce the over-approximation error by increasing the *precision* of the computations [4–6]. To avoid complete re-starts of the analysis upon parameter refinement for increased precision, some approaches use counterexample-guided refinements [7,8].

For applicability, it is also important to increase the *scalability* of these methods. A piece of work in this direction is [9], where the authors propose a scalable approach to compute the set of all states reachable by fixed-step simulation. Approaches like [10–12] decompose the state space into lower-dimensional subspaces in which reachability computations can be executed faster (but usually with less precision). One of the few parallelization approaches is presented in [13]; besides speeding up sequential computations, the authors propose to parallelize flowpipe computations for the over-approximation of reachability between two discrete state changes.

In this paper we propose a *parallelization* approach for a sequential algorithm [8], which applies flowpipe-construction-based reachability analysis in an iterative counterexample-guided refinement loop for error reduction. In contrast to [13] we do not parallelize the construction of a single flowpipe, but compute several flowpipes independently by parallel threads. An extension could additionally apply parallelization according to [13], but it is left for future work. Without a refinement loop, different flowpipe computations would be independent and thus their parallelization would be natural. However, our experience shows that achieving a work-load balance at low communication costs is challenging. Furthermore, the refinement loop makes additional synchronization necessary, which we keep at a minimum to reduce unnecessary synchronization costs. We implemented our method and provide some experimental results.

The rest of this paper is structured as follows: Sect. 2 contains preliminaries on flowpipe-construction-based reachability analysis and its embedding in a refinement loop as introduced in [8]. Section 3 presents our parallelization approach, followed by experimental results in Sect. 4. We conclude the paper in Sect. 5.

2 Preliminaries

2.1 Hybrid Automata

Hybrid automata are a well-established formalism for modeling hybrid systems.

Definition 1 (Hybrid automata: Syntax [14]). *A* hybrid automaton *is a tuple* $\mathcal{H} = (Loc, Var, Flow, Inv, Edge, Init)$ *with the following components:*

– *Loc is a finite set of* locations *or* control modes.

- $Var = \{x_1, \ldots, x_d\}$ *is a finite ordered set of real-valued* variables*; sometimes we use the vector notation* $x = (x_1, \ldots, x_d)$*. The number* d *is called the* dimension *of* $\mathcal{H}$*. By* $\dot{Var}$ *we denote the set* $\{\dot{x}_1, \ldots, \dot{x}_d\}$ *of dotted variables (which represent first derivatives during continuous evolution), and by* Var' *the set* $\{x'_1, \ldots, x'_d\}$ *of primed variables (which represent values directly after a discrete change). Furthermore, given a variable set* X*, let* $Pred_X$ *denote a set of predicates with free variables from* X*.*
- $Flow : Loc \to Pred_{Var \cup \dot{Var}}$ *specifies for each location its* flow *or* dynamics*.*
- $Inv : Loc \to Pred_{Var}$ *assigns to each location an* invariant*.*
- $Edge \subseteq Loc \times Pred_{Var} \times Pred_{Var \cup Var'} \times Loc$ *is a finite set of* edges (ℓ_1, g, r, ℓ_2) *with* source *location* ℓ_1*,* target *location* ℓ_2*,* guard g*, and* reset *function* r*.*
- $Init : Loc \to Pred_{Var}$ *assigns to each location an* initial *predicate.*

While the presented approach can be generalized, in this work we focus on *linear* hybrid automata, where $Pred_{Var}$ is the set of all conjunctions of linear equalities and inequalities over Var, $Flow$ assigns to each location a linear ordinary differential equation (ODE) system of the form $\dot{x} = Ax$ with some $A \in \mathbb{R}^{d \times d}$, and where reset functions on discrete transitions are defined by affine mappings $x' = Ax + b$ with $A \in \mathbb{R}^{d \times d}$ and $b \in \mathbb{R}^d$.

A *state* $, = (\ell, \nu)$ of a hybrid automaton consists of a location $\ell \in Loc$ and a variable valuation $\nu : Var \to \mathbb{R}$. We refer to a set of states with a common location ℓ and valuations from a set $\mathcal{V}$ by $(\ell, \mathcal{V}) = \{(\ell, \nu) \,|\, \nu \in \mathcal{V}\}$.

The state of a hybrid automaton can be changed either by time or by discrete steps. A *time step* $(\ell, \nu) \xrightarrow{t} (\ell, f(\nu, t))$ (also called *flow*) of length t models the passage of t time units: the control location remains unchanged and the variable values evolve continuously according to a solution f of the ODE system $Flow(\ell)$; the time step is enabled only if the invariant $Inv(\ell)$ is satisfied during the whole time step, i.e., by all $f(\nu, t')$ with $0 \leq t' \leq t$. A *discrete step* $(\ell, \nu) \xrightarrow{e} (\ell', \nu')$ (also called *jump*) models a discrete change of the control mode: it follows an edge $e = (\ell, g, r, \ell') \in Edge$ which is enabled (i.e., ν satisfies g and ν' satisfies $Inv(\ell')$), where ν' results from ν by applying the affine mapping specified by r. Note that hybrid automata are in general non-deterministic, as a time step and several jumps can be enabled at the same time.

An *execution* or *path* $\pi = \sigma_0 \xrightarrow{t_0} \sigma'_0 \xrightarrow{e_0} \sigma_1 \xrightarrow{t_1} \ldots$ is a (finite or infinite) sequence of alternating time and discrete steps, starting in an initial state $\sigma_0 = (\ell_0, \nu_0)$ such that ν_0 satisfies $Init(\ell_0)$. A state is called *reachable* if there is a finite path leading to it. Given a hybrid automaton $\mathcal{H}$ and subset T of its states, the *reachability problem* poses the question whether some state of T is reachable in $\mathcal{H}$.

2.2 Reachability Analysis Based on Flowpipe Construction

In this work we use a *bounded flowpipe-construction-based reachability analysis* method for linear hybrid automata. As the reachability problem for linear hybrid automata is in general undecidable, this approach computes over-approximations of bounded reachability (with upper bounds on the number of jumps and on the

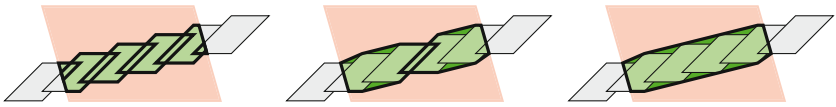


Fig. 1. Jump successors can be processed individually (6 sets on the left), clustered (2 sets in the middle) or aggregated (1 set on the right).

length of time steps). The computation starts from a set $(\ell_0, \mathcal{V}_0)$ of initial states and over-approximates, alternatingly, time successors within a *time horizon* T and jump successors iteratively up to a given *jump depth* J. As datatypes for state sets $\Omega = (\ell, \mathcal{V})$, different geometric or symbolic state set representations (e.g. boxes, convex polyhedra, zonotopes, support functions or Taylor models) can be used to over-approximate the valuation set $\mathcal{V}$ (when interpreted as a subset of $\mathbb{R}^d$).

To compute bounded time successors from a given set of valuations $\mathcal{V}$ in a location ℓ, the time horizon T is divided into N time segments of size $\delta = \frac{T}{N}$. For each $i = 0, \ldots, N-1$ the set of states reachable from $\mathcal{V}$ in ℓ within time $[i\delta, (i+1)\delta]$ is *over-approximated*, intersected with the invariant of ℓ and stored as a state set in Ω_i (called the ith *flowpipe segment*). The union $\bigcup_{i=0}^{N-1} \Omega_i$ of the flowpipe segments is referred to as the *flowpipe* and over-approximates the set of states reachable from $\mathcal{V}$ in ℓ within T time. If any of the flowpipe segments has a non-empty intersection with the set of unsafe states then the algorithm terminates (with an inconclusive answer due to over-approximation).

Otherwise, for each flowpipe segment $\Omega_i = (\ell, \mathcal{V}_i)$ and jump $e = (\ell, g, r, \ell')$ rooted in ℓ we determine an over-approximation $\Omega_{e,i}$ of the jump successors from Ω_i along e; this includes the intersection of $\mathcal{V}_i$ with g, the affine transformation of the result according to r, and the intersection with the invariant of ℓ'.

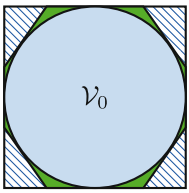


Fig. 2. Valuation set $\mathcal{V}_0$ over-approximated by a box (blue) and a convex polytope (green) [8]. (Color figure online)

One possibility is to apply the algorithm now iteratively to all non-empty jump successors $\Omega_{e,i}$ with $i = 0, \ldots, N-1$ and e being a jump leaving ℓ, until the jump depth has been reached. However, this approach is computationally very expensive. Alternatively, we can group the jump successors into a fixed number k of clusters (if there are more than k segments), over-approximate each cluster by one set, and continue the computations for each cluster over-approximation. If $k > 1$ then we call this procedure *clustering*, and for $k = 1$ we call it *aggregation* (see Fig. 1).

The choices of time segmentation, state set representation and clustering/aggregation parameters influence the over-approximation error. Usually, a smaller time step size δ, a more precise state set representation and finer clustering leads to a smaller error on the cost of increased computation time.

For instance, boxes require little computational effort for set operations but in general introduce more over-approximation error as e.g. convex polytopes do (see Fig. 2). We refer to [15,16] for further details.

During the analysis, we store the state sets for which flowpipes and jumps successors need to be computed in a *search tree*, whose depth is limited by the jump depth (see Fig. 3). The root node stores the initial states, whereas each other node n_i stores either the jump successor states of the flowpipe segment given by the parent, or a clustering/aggregation of such sets, depending on the parameter setting. If we label each parent-child connection with the union of the time intervals of the considered flowpipe segments and the jump taken, then the path from the root to a node describes a *symbolic path* $\Pi = I_0, e_0, \ldots, I_k, e_k$, which *represents* all paths $Paths(\Pi) = \{\sigma_0 \xrightarrow{t_0} \sigma_0' \xrightarrow{e_0} \sigma_1 \ldots \sigma_{l+1} \mid l \leq k \wedge \forall 0 \leq i \leq l.t_i \in I_i\}$. A path $\pi \in Paths(\Pi)$ that does not exist in the hybrid automaton is called *spurious*.

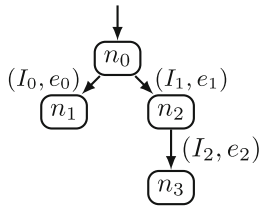


Fig. 3. An example for a search tree.

The structure of the search tree depends not only on the analyzed hybrid automaton but also on the analysis parameters. Non-determinism naturally causes a branching in the search tree, but over-approximation might cause not only larger sets in the nodes but also additional branching.

If the algorithm has terminated due to the detection of an unsafe state then the symbolic path to one of the nodes represents a *counterexample* path leading to an unsafe state. However, due to over-approximation, we do not know whether this counterexample is spurious or not.

2.3 Counterexample-Guided Parameter Refinement

Most available algorithms terminate at this point; the user needs to restart the search with adapted parameters to achieve a higher precision. To avoid a complete restart, in [8] we presented a counterexample-guided approach to repeat the search with refined parameters along (potentially spurious) counterexample paths only.

A user-defined collection of parameter settings is stored in an ordered list, the *refinement strategy*. We say that we compute at *refinement level* i when we use the $(i+1)$st setting in the refinement strategy. The refinement levels might differ e.g. in the state set representation, the time step size or in the clustering/aggregation settings.

The search starts at refinement level 0, i.e., with the first setting in the refinement strategy. When a potential counterexample is detected at refinement level i then we enforce an iterative re-computation of reachability within the counterexample's symbolic path Π (called the *refinement path*) at refinement level $i+1$ (unless i was the last defined level, in which case the algorithm terminates without any conclusive answer). These re-computations start at the

root node, for which the successors are computed at refinement level $i + 1$, however, only its successors along Π will be further processed by the refinement (i.e. only successors with symbolic path Π' for which $Paths(\Pi) \cap Paths(\Pi') \neq \emptyset$).

Note that several refinements might be applied to the same symbolic path. A special case is when a counterexample is detected *before* the whole previous counterexample has been refined, i.e., before reaching the end of the previous counterexample. In this case the counterexample must be spurious, because the previous over-approximative computations did not detect any unsafe states at that point; we continue the computations without additional refinement.

The refinements stop if either the counterexample could be shown to be spurious (path is safe) or we have tried all settings in the strategy but the potential counterexample could not be excluded. In the first case, the analysis continues with further successor computations; if the path with the spurious counterexample had less jumps than the jump depth, then also successors for its last state set are further processed, however, for these computations we jump back to refinement level 0.

Due to space restrictions, we cannot explain how we store the refined sets at all levels in a single search tree, and how we switch back from a higher refinement level to level 0 after the elimination of a spurious counterexample. Regarding the aspects of parallelization, it is not necessary to understand these mechanisms in detail. It is however important to notice that a node in the search tree can store several state sets, each computed at a different refinement level. Thus a node and a refinement level uniquely specify a state set stored in the tree.

3 Parallel Reachability Analysis

In the following we propose a parallelization approach for the previously introduced reachability analysis method with counterexample-guided parameter refinement for hybrid automata. We first discuss some aspects of a sequential implementation (Sect. 3.1) before we describe our parallel approach (Sect. 3.2) and implementation details (Sect. 3.3).

3.1 Sequential Analysis

In this section we recall from [8] some implementation-related concepts for the sequential analysis with counterexample-guided parameter refinement, as they will play basic roles for the parallelization.

Task. A task collects all information that is needed to compute flow and jump successors for a state set stored in a node of the search tree. In a classical approach without refinement, storing a reference to the search tree node would be sufficient for this purpose, assuming that all search parameters are globally accessible. With refinement, tasks are either *basic* at refinement level 0 or they are *refinement tasks* storing the refinement work at a positive refinement level for a node on a (potentially spurious) counterexample path. In both cases, the task

additionally needs to store the current refinement level (specifying the parameter values for the computations). In the case of refinement, the task also needs to store the symbolic path of the counterexample, to which also the refinement of the successors should be restricted.

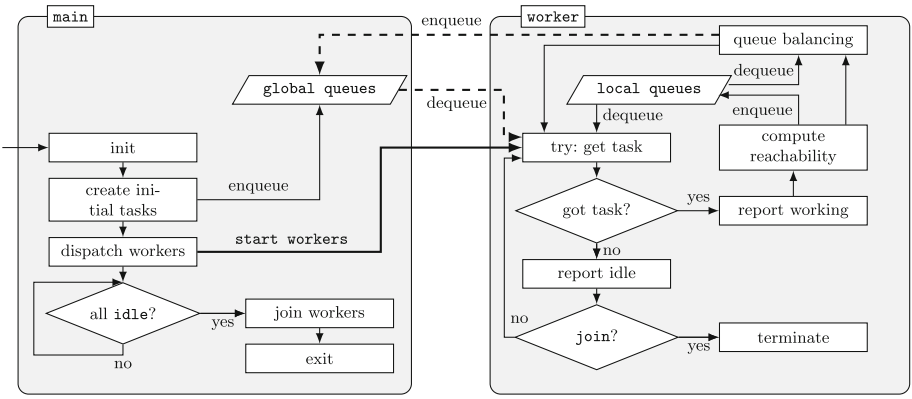


Fig. 4. HyDRA's execution structure. Dashed lines denote synchronized access.

Worker. A worker (or in the sequential setting *the* worker) is responsible for the execution of tasks. In our setup we employ one type of workers, which uses a state-of-the-art method for computing flowpipes and jump successors (see Sect. 2). However, as stated in [17], we could also consider specialized workers (e.g. applying different successor computation approaches dedicated to certain types of dynamics). We could even consider the decomposition of the state space as described in [10] and the invocation of specialized sub-workers on the components, but these ideas are not yet implemented.

Task Queue. Once a worker completed a basic task, it adds the corresponding jump successor state sets as new nodes to the search tree and creates tasks for them to trigger their processing (unless the jump depth has been reached or a potential counterexample has been detected). To keep track of the tasks that still need to be processed, in the sequential setup the worker maintains its own *task queue* – we will extend this concept for parallelization. In the implementation we use priority queues which allow to implement different search heuristics by modifying the order inside the queue.

Refinement Queue. Whenever a worker detects the potential reachability of some unsafe states, it triggers a refinement of the symbolic path to the current node (the *refinement path*) as presented in [8]. As counterexamples might share a prefix, when refining a node, the worker first checks whether the node has already been refined to the required level; if so then there will only be created requests for processing the children along the refinement path (in the form of new tasks).

For their storage, we want to prioritize refinement tasks over basic tasks. Instead of changing the ordering of the task queue, we do so by using a separate *refinement queue*, as it also allows for separate queue balancing methods (see Sect. 3.2).

3.2 Parallel Analysis

In this work we develop parallelization based on *multi-threading*. The tasks are natural units for parallel processing: multiple threads can implement workers (in a one-to-one correspondence between threads and workers) processing different tasks in parallel.

Local and Global Queues. As in the sequential case, each worker has a *local task queue* and a *local refinement queue*. Access to these local queues is restricted to the owning worker, therefore it does not require any synchronisation and is thus fast. Additionally, for work balancing, we need a mechanism to distribute tasks between threads. For this purpose we use a *global task queue* and a *global refinement queue*, which can be accessed by all workers in a synchronized fashion.

Initially there are some initial tasks (for initial state sets) in the global task queue, and the global refinement queue and all local queues are empty.

When idle, each worker tries first to obtain a task to process from its local refinement queue or its local task queue, in this order, to keep the synchronization overhead as small as possible. Only if both of its local queues are empty, the worker tries to obtain a task from the global refinement or the global task queue, using synchronized access. If both global queues are also empty, the worker re-checks the global queues regularly, until they are filled or until also all other local queues are empty, which leads to a synchronized completion of the algorithm.

If a worker processes a task, resulting new tasks will be added to the worker's local queues. I.e., without further balancing, the subtree under the currently processed node in the search tree will be analyzed by this worker only.

To allow work-balancing, workers can *move tasks* from their local queues to the corresponding global queues (from local task queue to global task queue, from local refinement queue to global refinement queue). We consider three heuristics for this balancing step, which apply after each completion of a task: (i) the worker pushes all but one tasks from its local queues to the global queues; (ii) only when the local queue size is larger than a certain threshold, tasks exceeding that threshold are moved from the local to the global queues; (iii) push a certain ratio of tasks from the local queues to the global queues. We expect that approaches (i) and (iii) will result in balanced work distribution at higher synchronization costs while approach (ii) should be better suited to limit these costs but lead to a less balanced execution. Note that the queue balancing happens after the completion of each task by a worker, i.e. when potential successor tasks have been added to the thread-local queues.

We also consider a different setting, where *only global queues* are present. In this setting, work is automatically distributed but getting work from the queues and adding new tasks to the queues require synchronization.

Node Synchronization. All workers share a single search tree. Without path refinement, the workers need to synchronize on the access to the global queues, but not on search tree nodes: each search tree node (below the jump depth level) will be referred to by exactly one task, which will be processed by exactly one worker. However, this is not the case for path refinement, as counterexample paths might share a prefix. To ensure thread-safety during path refinement, each worker first gets a lock on the tree node it intends to refine, processes the node, and gives the lock free before starting to process any other node.

3.3 Implementation

We extended our HyDRA tool by the presented approach, based on a previous implementation of the sequential counterexample-guided parameter refinement method. HyDRA uses the HyPro [15] `C++` library for state set representations, and it has been developed in a modular fashion to be easily extensible.

The general data flow of HyDRA is illustrated in Fig. 4. Similarly to the design principles presented in [17], the reachability analysis core (compute reachability) of HyDRA is built up of separate components dealing with the computation of continuous and discrete successors. Our implementation extends the existing concepts by distributing the analysis process among multiple threads.

The main thread of the tool is responsible for management operations e.g. invoking the parser, dispatching workers or plotting the computed reachability over-approximations, if required. Reachability is computed by a fixed number of separate worker-threads. After pre-processing and initialization by the main thread, i.e. parsing and creation of tasks from the initial states, the worker threads are created. Tasks are shared via globally accessible work queues – as stated before we maintain separate queues for refinement tasks and regular analysis tasks. Following the concept of work stealing, an idle worker with empty local queues obtains its next task to work on from a global work queue and processes it. Each worker extends the shared search tree by jump successor state sets and creates the corresponding new tasks for the work queue.

Signaling. Inter-thread communication is necessary to join workers after completion of the analysis. A worker reports idleness via an event system whenever there is no task in its local queues and no task available in the global queues. During idling, the worker repeatedly tries to get a task from the global queues; if this succeeds, the worker signals the end of its idling period (this signalling happens inside the synchronized access to the global queues). When all workers reported idleness i.e. all queues are empty, the main thread signals the worker threads to terminate. All workers are joined and post-processing of the computed sets e.g. plotting (in the figure: *exit*) can be performed. As signaling requires synchronization, the number of signals should be limited as far as possible.

Queue Access. To reduce the overhead introduced by synchronization, we equip our global queues with synchronized as well as non-synchronized methods for access. Idle workers can utilize non-synchronized methods for the global queues

to check for emptiness and only use synchronized access methods whenever the queue is not empty (after a second, synchronized check for emptiness while holding the lock for the queue). This allows to avoid unnecessary synchronization in scenarios where there are many idle workers constantly accessing the global queues which are empty most of the time but at the same time ensures that dequeuing of tasks is still synchronized.

Thread-Safe Linear Optimization. Despite synchronized access to the task queues and the single search tree nodes, adjustments to the implemented state set representations in HyPro have to be considered to make the tool thread safe. In general this does not require specialized approaches, however adapting an embedded linear optimization engine required some effort. HyPro allows to use different linear solving backends with a fallback to GLPK which are wrapped into an optimizer class. It is known that GLPK is not thread-safe, however with minor modifications it is possible to obtain a re-entrant version. This can be achieved by changing the maintained global GLPK-context object to a thread-local context. Now that each thread maintains its own GLPK-context, special care has to be taken to avoid memory-leaks. We extend our optimization wrapper class by mapping the unique thread id to the corresponding GLPK context and its problem instances. State set representations (e.g. support functions) which hold their own optimizer class instance now have to make sure the GLPK context for this instance is properly deleted upon joining threads, as for every thread which accesses this state set the corresponding mapping in the optimizer class is extended. To avoid this we provide clean-up methods, which should be called before a thread is joined. Clean-up deletes all GLPK-problem instances and removes the thread-local GLPK-context instance (which can only be deleted by its creating thread). In general creating a GLPK-problem instance upon request and deleting it afterwards would solve this issue as well – however as the same problem instance usually is used several times we reduce the overhead of creating and deleting these instances by keeping them as long as possible.

4 Experimental Results

We tested our implementation on several well-known benchmarks with a timeout (denoted as *to*) of 10 min on a machine equipped with 48×2.1 GHz AMD Opteron CPUs and a memory limit (*mo*) of 8 GB.

Benchmarks. Three benchmarks have been selected for empirical evaluation. We include two instances of the navigation benchmark [18] – instance 9 (`na09`, time horizon $T = 3$ s, jump depth $J = 9$) and instance 11 (`na11`, time horizon $T = 3$ s, jump depth $J = 8$). Both instances model a point mass moving on a two-dimensional plane subdivided into cells which each model different acceleration affecting the movement of the point mass. Due to the large set of initial states, these benchmarks usually exhibit strong branching behavior and thus

should be well-suited to evaluate the capabilities of our implementation. Furthermore, we include an instance of Fisher's mutual exclusion protocol benchmark (`fish`, $T = 12\,\mathrm{s}$, $J = 13$) which also was used in [19]. This benchmark models several processes competing for a shared resource which can be accessed in a mutually exclusive way. As all processes have the same priority the model is non-deterministic and we expect to obtain a shallow search tree during analysis. In our evaluation we did not include benchmarks with little non-deterministic choices. Our central goals are to investigate on the influence of queue balancing methods and on the potential speed-up which can be achieved by parallelization. Additionally, our results show that the overhead caused by synchronization is small (see below) so we can expect little influence on running times for benchmarks with little branching.

Table 1. Parameter settings: Refinement strategies are lists of configurations, each configuration specified by a triplet (1) state set representation (box, support functions (sf)), (2) time step size, (3) aggregation (agg)/clustering in k clusters ($cl.k$). Additionally, the last column specifies the queue balancing rate.

Name	Refinement strategy	Work balancing
s_0	$(box, 0.1, agg), (sf, 0.01, agg), (sf, 0.001, agg)$	100%
s_1	$(sf, 0.1, agg), (sf, 0.01, agg)$	100%
s_2	$(sf, 0.1, agg), (sf, 0.01, cl.5)$	100%
s_3	$(box, 0.1, agg), (box, 0.01, agg)$	100%
s_4	$(box, 0.1, agg), (box, 0.01, cl.3)$	100%
s_5	$(box, 0.1, agg), (box, 0.01, cl.3)$	10%
s_6	$(box, 0.1, agg), (box, 0.01, cl.3)$	50%
s_7	$(box, 0.1, agg), (box, 0.01, cl.3)$	Global queue only

Settings. For our experiments we consider 8 different settings (see Table 1). Even though path refinement is not the main focus of our presented approach, all 8 settings support path refinement as this involves synchronization (see Sect. 3).

Each setting specifies a refinement strategy and a work queue balancing heuristics. A refinement strategy is a sequence of triplets, each triplet specifying (1) the state set representation used, (2) the time step size for flowpipe construction and (3) settings for aggregation/clustering. In regard to queue balancing, we made experiments with pushing all tasks above a threshold from the local queues to the global queues, but this was far less stable in efficiency than pushing a certain percentage of the local queue contents, therefore here we include only experiments with the latter. In Table 1, the work queue balancing heuristics specifies which portion of the local queues is moved to the global queues after the completion of each task (at least one task is always left in non-empty local queues, i.e., 100% means all but one).

Settings s_0–s_4 differ in their refinement heuristics, but they are all eager in pushing all but one task from the local to the global queues after the completion of each task. Contrary, settings s_4–s_7 share the same refinement heuristics but they differ in their work balancing method. Especially, setting s_7 completely avoids thread-local queues: every worker operates on the global queues directly. The difference is that, while in all other settings the work distribution takes place at the end of the flowpipe computation in a batch, s_7 pushes single successor tasks to the global queues during its computations such that idle workers potentially could start computation earlier. As the experimental results will show, this works surprisingly good, even though the increased synchronization effort is recognizable.

Table 2. Running times [sec.] for settings s_0–s_7, timeout (to) = 10 min, memout (mo) = 8 GB, † = safety cannot be shown. Running times averaged over 10 runs.

Benchmark	Setting	#threads						
		1	2	4	8	16	32	48
`na09`	s_0	**21.99**	20.32	20.32	20.40	20.34	20.29	20.35
	s_1	24.87	**15.72**	**11.87**	**11.70**	**11.68**	11.70	11.72
	s_2	to	to	to	to	mo	mo	mo
	s_3	†	†	†	†	†	†	†
	s_4	263.8	134.9	69.34	36.87	21.68	16.70	15.63
	s_5	252.8	127.9	64.79	32.85	17.00	**10.41**	**7.51**
	s_6	263.5	132.8	68.70	36.20	20.90	15.53	13.95
	s_7	78.52	46.60	32.01	29.52	34.21	42.23	45.03
`na11`	s_0	70.49	45.72	45.39	45.42	45.41	45.47	45.44
	s_1	**18.47**	**9.81**	**6.15**	**5.03**	**4.68**	4.49	4.50
	s_2	to	290.7	146.4	75.53	39.92	22.45	16.50
	s_3	†	†	†	†	†	†	†
	s_4	95.73	47.05	24.04	12.21	6.42	**3.60**	3.13
	s_5	93.68	45.85	23.28	12.03	6.57	4.02	3.54
	s_6	92.11	47.16	24.02	12.20	6.62	3.74	**3.02**
	s_7	95.92	49.12	25.12	13.03	8.02	6.16	6.49
`fish`	s_0	40.66	20.46	**10.43**	**5.49**	**2.96**	1.84	**1.61**
	s_1	to	to	to	393.9	201.5	107.2	79.02
	s_2	to	to	to	394.3	201.4	107.4	79.07
	s_3	40.57	20.44	10.47	5.54	2.97	**1.82**	1.78
	s_4	**40.56**	20.45	10.49	5.55	2.97	1.79	1.83
	s_5	40.63	20.47	10.87	6.76	4.56	3.92	3.96
	s_6	40.67	**20.42**	10.47	5.53	**2.96**	1.84	1.70
	s_7	42.73	21.79	11.26	6.06	3.68	3.45	3.93

Results. The running times for our experiments are listed in Table 2. In general, we can observe a speed-up when increasing the number of worker threads – we could achieve a speedup of up to factor 33 (`na09`) which in this case results in ∼70.1% efficiency ($efficiency = \frac{speedup}{\#threads}$) of the parallelization (`na11`: max. factor 30, `fish`: max. factor 25). Even though a general speed-up when using more worker threads can be observed, some instances (e.g. `na09`, s_0) stabilize in their running times. This indicates that either work is not well balanced or there is a heavy synchronization overhead.

For interpreting the results, it is important to mention that processing each single task is in general computationally expensive: the time required to compute a flowpipe is usually long in comparison to the time it takes to acquire a lock for synchronization and move tasks to global queues. Consequently the running times using one thread in our implementation resemble the running times of a purely sequential approach. Furthermore, with aggregation/clustering the number of generated new tasks is often relatively small. For example, for a deterministic system a task might generate just a single successor task, in which case no work balancing would take place at all. This might lead to insufficient work balancing and explain why for some benchmarks and some settings involving more workers does not lead to any additional speedup.

To further investigate upon this we ran the benchmarks with up to 48 threads. For benchmark instances such as the navigation benchmark in combination with settings where aggregation was used (s_0, s_1, s_3) we can observe that the running times already converge for a low number of threads as there are not enough tasks created during analysis such that most threads idle. The running times for these settings do not significantly increase when using more threads which confirms that our implementation successfully minimizes the synchronization effort required. An exception is setting s_7 on benchmark `na09`, where the running times increase when using more than 8 threads; as this setting only uses global queues, the increased need for synchronization is reflected in the running times.

To investigate on the actual work distribution we collected the number of tasks processed by each worker thread. Table 3 shows the coefficient of variation (CV) of these results to allow for statements about variance in the work distribution. The coefficient of variation as a relative measure for variance gives the influence of the variance of data on the mean in percent. Lower percentages hereby indicate a lower variance in data.

We can observe the influence of different queue balancing methods for benchmarks with settings which produce a lot of tasks (s_4–s_7). With increasing number of threads the average number of processed tasks per worker decreases. When using settings which produce too few tasks, many worker threads idle, thus increasing the variance of processed tasks per worker (see e.g. `na09`, s_0). As expected the setting using only global queues shows the lowest CV throughout the experiments as all available tasks are immediately shared.

Settings with local queues where 100% of the created tasks are shared are expected to exhibit a similar CV as when using global queues only, there are only two differences: firstly, when using global queues only, tasks are shared

Table 3. Coefficient of variation (left) and idle time (right) in percent for settings s_0–s_7, "–" marks failures (timeout, memout). Unsuccessful settings are left out.

Benchmark	Setting	#threads						#threads					
		2	4	8	16	32	48	2	4	8	16	32	48
na09	s_0	87.5	85.4	102.2	133.5	197.2	220.6	18.72	34.96	36.52	33.5	15.28	12.2
	s_1	32.7	43.6	39.0	44.8	92.5	118.4	10.63	28.78	36.52	28.29	12.76	10.21
	s_4	**0.1**	0.7	1.0	**1.3**	**1.7**	**2.2**	**0.04**	**0.18**	0.44	0.85	1.09	1.22
	s_5	0.4	1.1	1.8	2.7	4.0	4.9	0.16	0.46	1.07	2.30	4.33	6.16
	s_6	0.2	0.4	1.0	1.4	1.8	**2.2**	0.05	**0.18**	0.45	0.86	1.33	1.69
	s_7	0.4	**0.6**	**0.9**	**1.3**	2.2	2.7	0.11	0.23	**0.30**	**0.41**	**0.38**	**0.41**
na11	s_0	45.3	44.3	70.4	130.8	175.1	215.8	7.52	6.05	3.54	2.44	1.36	0.74
	s_1	24.0	15.3	29.2	45.1	90.5	121.0	4.13	20.95	40.91	47.22	33.84	25.93
	s_2	**0.4**	**0.9**	**1.9**	3.6	6.0	7.6	0.11	0.51	2.33	5.76	12.10	17.2
	s_4	0.9	1.7	10.3	11.0	15.7	13.5	0.11	0.44	1.21	3.45	5.79	6.17
	s_5	2.2	3.2	5.3	8.8	13.6	16.2	0.17	0.64	1.43	3.82	6.62	7.75
	s_6	1.4	2.1	2.6	17.3	11.9	12.6	0.07	0.46	0.81	3.35	6.35	7.74
	s_7	2.5	3.0	3.6	**3.3**	**3.9**	**5.5**	**0.01**	**0.30**	**0.72**	**1.63**	**2.67**	**2.78**
fish	s_0	0.6	3.6	7.6	8.8	11.6	14.3	0.32	1.22	3.32	5.94	10.21	12.33
	s_1	–	–	6.8	8.0	10.0	13.2	–	–	**0.44**	**1.09**	**2.44**	3.7
	s_2	–	–	7.6	8.5	10.0	12.7	–	–	**0.44**	**1.06**	**2.65**	3.8
	s_3	0.8	3.3	7.4	9.3	11.8	13.9	0.29	1.43	3.84	5.88	10.45	11.37
	s_4	0.8	3.4	7.1	8.0	11.3	13.9	0.29	1.19	4.39	6.41	9.90	11.70
	s_5	**0.3**	2.6	14.9	24.8	67.6	99.8	0.24	2.00	1.96	7.73	15.30	14.81
	s_6	0.9	3.4	8.1	8.4	11.6	14.0	0.32	1.29	3.98	6.01	10.42	11.85
	s_7	0.5	**1.2**	**2.6**	**2.9**	**4.1**	**4.9**	**0.23**	**0.67**	1.44	2.56	2.83	**2.50**

immediately after their creation, whereas in the presence of local queues sharing happens after task completion; secondly, 100% sharing with local queues is not exactly 100% as one single task is kept for further processing in a local queue. Strategies where a worker only shares part of its created tasks (s_5, s_6) show a larger variance i.e. work is less equally distributed. With regard to the observed running times we can deduce that sharing work comes at a price – even though setting s_7 has the lowest variance, the running times in comparison to settings s_4–s_6, which share the same analysis parameters are worse.

Note that a low CV can also be achieved when many threads are taking turns in processing a small number of such tasks. Therefore, we also analyzed the average share of idle time for all threads (see Table 3, right). We can conclude that the increased running time for setting s_7 indeed can be amounted to synchronization, as the idle time for the workers is amongst the lowest ones.

5 Conclusion

We have presented a natural approach to parallelize reachability analysis for linear hybrid systems. Experimental results show a general reduction of the analysis times. The observed synchronization overhead is minor compared to what

we gain from the parallel execution and thus this approach is usable also for systems for which the search tree has a low level of branching (e.g. for deterministic systems). Naturally, the possibilities of work sharing are restricted to problem instances with a low level of non-determinism. The used modular approach allows for several extensions and improvements as future work: (i) combining this method with the approach presented in [13], and (ii) using specialized workers which allow for subset-computations which can be performed in parallel.

References

1. Althoff, M., Dolan, J.M.: Online verification of automated road vehicles using reachability analysis. IEEE Trans. Robot. **30**(4), 903–918 (2014)
2. Frehse, G., et al.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_30
3. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_18
4. Girard, A.: Reachability of uncertain linear systems using zonotopes. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 291–305. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31954-2_19
5. Frehse, G., Kateja, R., Le Guernic, C.: Flowpipe approximation and clustering in space-time. In: Proceedings of HSCC 2013, pp. 203–212. ACM (2013)
6. Le Guernic, C., Girard, A.: Reachability analysis of linear systems using support functions. Nonlinear Anal. Hybrid Syst. **4**(2), 250–262 (2010)
7. Bogomolov, S., Donzé, A., Frehse, G., Grosu, R., Johnson, T.T., Ladan, H., Podelski, A., Wehrle, M.: Guided search for hybrid systems based on coarse-grained space abstractions. STTT **18**(4), 449–467 (2016)
8. Schupp, S., Ábrahám, E.: Efficient dynamic error reduction for hybrid systems reachability analysis. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 287–302. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_17. Accessible for reviewers under https://ths.rwth-aachen.de/research/publications/
9. Bak, S., Duggirala, P.S.: Simulation-equivalent reachability of large linear systems with inputs. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 401–420. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_20
10. Schupp, S., Nellen, J., Ábrahám, E.: Divide and conquer: variable set separation in hybrid systems reachability analysis. In: Proceedings of QAPL 2017. EPTCS, vol. 250, pp. 1–14. Open Publishing Association (2017)
11. Bogomolov, S., Forets, M., Frehse, G., Podelski, A., Schilling, C., Viry, F.: Reach set approximation through decomposition with low-dimensional sets and high-dimensional matrices. CoRR abs/1801.09526 (2018)
12. Chen, X., Sankaranarayanan, S.: Decomposed reachability analysis for nonlinear systems. In: Proceedings of RTSS 2016, pp. 13–24. IEEE Computer Society Press (2016)
13. Ray, R., Gurung, A.: Parallel state space exploration of linear systems with inputs using XSpeed. In: Proceedings of HSCC 2015, pp. 285–286. ACM (2015)

14. Henzinger, T.A.: The theory of hybrid automata. In: Proceedings of LICS 1996, pp. 278–292. IEEE Computer Society Press (1996)
15. Schupp, S., Ábrahám, E., Makhlouf, I.B., Kowalewski, S.: HyPro: A C++ library of state set representations for hybrid systems reachability analysis. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 288–294. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_20
16. Schupp, S., Ábrahám, E., Chen, X., Ben Makhlouf, I., Frehse, G., Sankaranarayanan, S., Kowalewski, S.: Current challenges in the verification of hybrid systems. In: Berger, C., Mousavi, M.R. (eds.) CyPhy 2015. LNCS, vol. 9361, pp. 8–24. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25141-7_2
17. Frehse, G., Ray, R.: Design principles for an extendable verification tool for hybrid systems. In: Proceedings of ADHS 2009, pp. 244–249. IFAC-PapersOnLine (2009)
18. Fehnker, A., Ivančić, F.: Benchmarks for hybrid systems verification. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 326–341. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24743-2_22
19. Bu, L., Ray, R., Schupp, S.: ARCH-COMP17 category report: bounded model checking of hybrid systems with piecewise constant dynamics. In: Proceedings of ARCH 2017. EPiC Series in Computing, vol. 48, pp. 134–142. EasyChair (2017)

FastLane Is Opaque – a Case Study in Mechanized Proofs of Opacity

Gerhard Schellhorn[1], Monika Wedel[2], Oleg Travkin[2], Jürgen König[2(✉)], and Heike Wehrheim[2]

[1] Universität Augsburg, Augsburg, Germany
schellhorn@informatik.uni-augsburg.de
[2] Paderborn University, Paderborn, Germany
{oleg82,jkoenig,wehrheim}@uni-paderborn.de

Abstract. Software Transactional Memory (STM) algorithms provide programmers with a high-level synchronization technique for concurrent programming. STMs guarantee "seemingly atomic" access to shared state via transactions. This seeming atomicity is the standard requirement on STM implementations and formalized in the concept of *opacity*. The standard proof technique for opacity is via *refinement*: the STM implementation is shown to refine an IO automaton called TMS2 which itself is known to be opaque.

This paper presents a case study of proving opacity via TMS2 refinement. Our case study concerns the FastLane implementation of STM which is specifically designed to achieve good performance on varying contention: it supports different modes for low and high thread counts plus provides a switching scheme between modes. This basic concept provides new challenges for verification: besides having to prove opacity of every mode itself, we also need to show that switching does not invalidate opacity. For both parts, we present fully mechanized proofs of opacity carried out in the interactive theorem prover KIV.

1 Introduction

Software Transactional Memory (STM) as proposed by Shavit and Touitou [23] has been introduced as a high-level synchronization technique for concurrent access to shared state. STM provides programmers with the concept of *transactions*, and thereby replaces standard synchronization primitives used for achieving mutual exclusion like locks or semaphores. A transaction is expected to perform as if it was being executed *atomically*. STM implementations need to guarantee this seeming atomicity while at the same time – for performance reasons – allowing for concurrency. Typically, STMs achieve this via fine-grained synchronization primitives (like compare-and-swap operations).

Opacity [14] has recently evolved as the standard correctness condition for STMs formalizing this seeming atomicity. In its basic flavour, opacity is similar to serializability of database transactions [22]: Each concurrent execution of transactions (i.e., an execution in which read and write operations of different

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 105–120, 2018.
https://doi.org/10.1007/978-3-319-92970-5_7

transactions are interleaved) has to be equivalent to a sequential one (i.e., one in which operations of different transactions are never interleaved). In addition, opacity has specific requirements on aborting transactions in that all transactions, even aborting ones, need to see a consistent view of the shared state as produced by prior successfully committed transactions.

Verifying opacity of an STM algorithm typically proceeds by a proof of *refinement*. To this end, Doherty et al. [9] have proposed an (input-output)-automaton called TMS2 which serves as the abstract specification in the refinement proof. TMS2 has been shown to be opaque [9,18] and in addition is given in an operational style which allows for refinement proofs by simulation. A large number of opacity proofs have recently been done in this form (e.g. [2,6,8,17]).
In this paper, we provide another case study of proving opacity. The case study concerns the FastLane STM algorithm of Wamhoff et al. [24]. FastLane is designed to work specifically well on varying contention on access to the shared state. It provides different *modes* for different thread counts as to avoid the overhead in synchronization when not necessary. The mode is dynamically adapted to the number of transactions. This dynamic mode switching provides new challenges for verification: in addition to verifying opacity of every mode in isolation, we also need to prove that mode switching does not invalidate opacity. To this end, we provide (1) a general definition of an (input-output) automaton switching being two automata and (2) proof obligations guaranteeing overall refinement of a mode-switching automaton (wrt. some specification A) when composed out of automata refining A. We prove soundness of this general theorem, and then prove that the resulting proof obligations are satisfied by the combined implementation with mode switching, implying that it refines TMS2 and is therefore opaque. All proofs are fully mechanized within the interactive theorem prover KIV [10] and available online [11].

2 The FastLane Algorithm

Our chosen case study is the FastLane algorithm by Wamhoff et al. [24]. It suggests three operational modes in order to dynamically change the STM algorithm based on the number of processes running transactions. For one process, no synchronization is necessary as all transactions will execute sequentially. For this case, a simple *sequential* STM algorithm is sufficient. When the number of processes is high, an *STM algorithm* has to take into account all sorts of conflicts. Well scaling algorithms usually add some bookkeeping in order to manage concurrency of transactions [5,7,15]. However, when there are only few threads available, the bookkeeping part of an algorithm becomes an avoidable overhead. The FastLane mode itself is dedicated to particularly this third scenario, and can improve throughput of transactions compared to algorithms that are designed for a high number of concurrent transactions. Note that the name "FastLane" sometimes refers to the algorithm with all three modes and sometimes to just the mode for low numbers of threads.

The FastLane algorithms allows for switching modes whenever the current execution is in a *quiescent* state, i.e., when no transaction is pending. It may

seem as if it is sufficient to show correctness of each algorithm separately in such a case, because each algorithm operates in isolation from the other. However, it is their composition that we want to show to be correct and hence it has to be taken into consideration.

2.1 The FastLane Mode

Before looking at the specific algorithm, we first give some more general explanation on STMs. An STM algorithm provides the concept of *transactions* to programmers. Transactions access locations in shared state (in the algorithm given by parameter `addr`). The following operations need to be supplied by STMs:

- a *start* operation (sometimes also called *begin* operation),
- a *write* operation writing a particular value on a particular location,
- a *read* operation reading from a particular location,
- a *cancel* and a *commit* operation.

These operations are sometimes specialized for specific transactions, e.g., an STM might have read-only transactions with optimized read operations.

Transactions furthermore need to be *well-formed*, i.e., always start with the start operation, then to be followed by a number of reads and writes and finally end with a call to the commit operation. The necessary synchronization to get "seeming atomicity" is achieved via the use of *meta-data* which e.g. might record specific accesses of transactions (like read and write sets). Basically, the STM has to make sure that every transaction operates on a consistent view of the shared state.

The FastLane mode is designed to operate in a concurrent setting with low contention. It assigns exactly one thread the role of being a *master*. All other threads become *helper* threads. The master thread has priority over the other threads and is therefore on the "fast lane".

Figure 1 describes the algorithm of the FastLane mode. At the start of a transaction, it is determined whether the current thread is a master or a helper (lines L4–14). Before this check, the `master` lock is acquired and if the current thread is the master, the `master` lock is kept throughout the whole transaction until it commits. The `master` lock must be acquired before the check, because helper threads can also become master threads, if the transaction is supposed to be performed pessimistically (L9–12). The latter protects transactions from being aborted due to conflicts with other transactions, because only helper transactions can abort in this mode.

A master transaction (L15–29) is straightforward for the most part. The shared variable `cntr` acts as an indicator for a writing transaction. If `cntr` is odd, then there is currently a writing transaction (otherwise not). On the first write, the master sets `cntr` to an odd value (L21–22). In any case, the location `addr` to which it writes is marked as dirty[1]. Helper transactions use the `dirty`

[1] A hash function reduces the size of the dirty array, but can also cause transactions to abort due to hash collisions when there is actually no conflict.

```
1  function START(pessimistic)
2    if masterID = threadID then
3      ttas-lock(master)
4      if masterID = threadID then
5        MASTERSTART
6      else
7        ttas-unlock(master)
8        HELPERSTART
9    else if pessimistic then
10     ttas-lock(master)
11     masterID := threadID
12     MASTERSTART
13   else
14     HELPERSTART

15 function MASTERSTART
16
17 function MASTERREAD(addr)
18   return *addr
19
20 function MASTERWRITE(addr, val)
21   if (not cntr & 1) then
22     cntr := cntr + 1
23   dirty[hash(addr)] := cntr
24   *addr := val
25
26 function MASTERCOMMIT
27   if cntr & 1 then
28     cntr := cntr + 1
29   ttas-unlock(master)

30 function VALIDATE
31   if cntr <= start then
32     return true
33   for addr in (rd-set ∪ wr-set) do
34     if dirty[hash(addr)]> start then
35       return false
36   return true

37 function HELPERSTART
38   setjmp(ctxt)
39   start := cntr &∼ 1
40
41 function HELPERREAD(addr)
42   if contains(wr-set, addr) then
43     return GET(wr-set, addr)
44   val := *addr
45   if dirty[hash(addr)]>start then
46     ABORT
47   ADD(rd-set, addr)
48   return val
49
50 function HELPERWRITE(addr, val)
51   if dirty[hash(addr)]>start then
52     ABORT
53   PUT(wr-set, addr, val)
54
55 function HELPERCOMMIT
56   if EMPTY(wr-set) then
57       return
58     mcs-lock(helpers)
59     ttas-lock(master)
60     if(not VALIDATE) then
61       ttas-unlock(master)
62       mcs-unlock(helpers)
63       ABORT
64     cntr := cntr + 1
65     foreach(addr, val) in wr-set do
66       dirty[hash(addr)] := cntr
67       *addr := val
68     cntr := cntr + 1
69     ttas-unlock(master)
70     mcs-unlock(helpers)
71
72   function ABORT
73     CLEAR(rd-set, wr-set)
74     longjmp(ctxt)
```

Fig. 1. The FastLane algorithm as proposed by Wamhoff et al. [24]

array to detect conflicts with the master transaction and abort in this case. On commit (L26–29), the master increments `cntr` back to an even value and releases its `master` lock, which enables writing helper transactions to commit.

Helper transactions (L37–74) are more complicated as they contain all the handling of conflicts with the master thread. It is worth noting that helpers restart immediately after being aborted (L74). Therefore, the first thing a helper does is to set a target jump location (L38). The second task of a helper is to fetch the latest even value of `cntr` and store it in its `start` variable (L39). The variable `start` is used to determine, whether the helper attempts to read from or write to a dirty location (L45–46 and L51–52) and causes the transaction to abort and reset, if this is indeed the case. Besides this, a helper transaction has sets for locations read and written (`rd-set` and `wr-set`). During read attempts, a helper prioritizes its writes from the `wr-set` over the shared memory (L42–48) as `wr-set` contains the later entries. On the other hand, write attempts do not actually write to memory, but add the pair of location and value to the write set `wr-set`. The actual writing appears during the transaction's commit (L55–70). If the write set of a helper transaction is empty, it can safely return

(L56–57), because conflicts with other transactions would have been detected during its read attempts (L45–46). However, if `wr-set` is not empty, the helper must acquire the `helpers` and the `master` locks first (L58–59). The `helpers` lock protects it from conflicts with other helpers, while acquiring the `masters` locks ensures that there is currently no master transaction. After acquiring both locks, the helper checks once again for conflicts with other transactions by checking the state of the `dirty` array for each locations in its read and write set (L30–36). This validation is necessary, because previously read locations might have been overwritten by other transactions at this stage. If the validation fails, the transaction releases both locks and aborts. Otherwise, it continues with incrementation of `cntr` to an odd value, followed by writing each write from the `wr-set` to the memory and by setting its `dirty` value to the new `cntr` value. The helper transaction completes its executions successfully with a second increment of `cntr` to an even value and the release of both previously acquired locks.

3 Proof Method

We are ultimately interested in showing the opacity [14] of the FastLane mode as well as its composition with the other modes. Opacity [14] is the standard correctness criterion for transactional memory. We will do so by showing FastLane and the composition to refine an abstract, operational specification called TMS2 [9]. TMS2 is known to be opaque [9,18]. Hence, every refinement of TMS2 is opaque as well. We thus never define opacity in this paper and only work with TMS2 in the following. TMS2 is modelled as an input/output automaton (*IO automaton* or IOA). Our definition of IO automata follows that of Lynch and Tuttle [19][2]. The idea in IOAs is to distinguish between *internal* and *external* actions, where the external actions of a system are visible to the environment and the internal ones are hidden from it.

Definition 1. *An* IO automaton A *consists of*

- *a set of states* $states(A)$ *and a set of initial states* $start(A) \subseteq states(A)$,
- *a set of actions* $act(A) = int(A) \dot\cup ext(A)$, *partitioned into internal and external ones, and*
- *a transition relation* $steps(A) \subseteq states(A) \times act(A) \times states(A)$.

Figure 2 shows the specification of TMS2 in a generic format we use for all automata A. The states $states(A)$ are given as a tuple of typed state variables. Constraints on initial values describe initial states. The transition relation $steps(A)$ is given by deterministic steps for every action a. A boolean preconditions $Pre_a(s)$ specifies when the action is enabled and an effect function Eff_a: $states(A) \to states(A)$ that is defined by parallel updates on the state variables gives the new state. We have $steps(A)(s, a, s') \Leftrightarrow Pre_a(s) \wedge s' = Eff_a(s)$.

[2] We leave out the equivalence relation on locally controlled actions, since we do not need it for our refinement proofs.

State variables:
$memories : \mathbb{N} \nrightarrow L \rightarrow V$, initially $\mathrm{dom}(memories) = \{0\} \wedge initMem(memories(0))$
$pc_t : PC$, initially $pc_t = \mathrm{notStarted}$ for all $t \in T$
$beginIdx_t : \mathbb{N}$, initially unconstrained for all $t \in T$
$rdSet_t : L \nrightarrow V$, initially empty for all $t \in T$
$wrSet_t : L \nrightarrow V$, initially empty for all $t \in T$

Transition relation:

$inv_t(\texttt{TMBegin})$
Pre: $pc_t = \mathrm{notStarted}$
Eff: $pc_t := \mathrm{beginPending}$
$beginIdx_t := maxIdx$

$resp_t(\texttt{TMBegin})$
Pre: $pc_t = \mathrm{beginPending}$
Eff: $pc_t := \mathrm{ready}$

$inv_t(\texttt{TMRead}(l))$
Pre: $pc_t = \mathrm{ready}$
Eff: $pc_t := \mathrm{doRead}(l)$

$resp_t(\texttt{TMRead}(v))$
Pre: $pc_t = \mathrm{readResp}(v)$
Eff: $pc_t := \mathrm{ready}$

$inv_t(\texttt{TMWrite}(l, v))$
Pre: $pc_t = \mathrm{ready}$
Eff: $pc_t := \mathrm{doWrite}(l, v)$

$resp_t(\texttt{TMWrite})$
Pre: $pc_t = \mathrm{writeResp}$
Eff: $pc_t := \mathrm{ready}$

$inv_t(\texttt{TMEnd})$
Pre: $pc_t = \mathrm{ready}$
Eff: $pc_t := \mathrm{doCommit}$

$resp_t(\texttt{TMEnd})$
Pre: $pc_t = \mathrm{commitResp}$
Eff: $pc_t := \mathrm{committed}$

$inv_t(\texttt{cancel})$
Pre: $pc_t = \mathrm{ready}$
Eff: $pc_t := \mathrm{cancelPending}$

$resp_t(\texttt{abort})$
Pre: $pc_t \notin \{\mathrm{notStarted}, \mathrm{ready}, \mathrm{commitResp}, \mathrm{committed}, \mathrm{aborted}\}$
Eff: $pc_t := \mathrm{aborted}$

$\texttt{DoCommitReadOnly}_t(n)$
Pre: $pc_t = \mathrm{doCommit}$
$\mathrm{dom}(wrSet_t) = \varnothing$
$validIdx(t, n)$
Eff: $pc_t := \mathrm{commitResp}$

$\texttt{DoCommitWriter}_t$
Pre: $pc_t = \mathrm{doCommit}$
$readCons(latestMem, rdSet_t)$
Eff: $pc_t := \mathrm{commitResp}$
$memories := memories \cup \{maxIdx + 1 \mapsto (latestMem \oplus wrSet_t)\}$

$\texttt{DoRead}_t(l, n)$
Pre: $pc_t = \mathrm{doRead}(l)$
$l \in \mathrm{dom}(wrSet_t) \vee validIdx(t, n)$
Eff: if $l \in \mathrm{dom}(wrSet_t)$ then
$pc_t := \mathrm{readResp}(wrSet_t(l))$
else
$v := memories(n)(l)$
$pc_t := \mathrm{readResp}(v)$
$rdSet_t := rdSet_t \oplus \{l \rightarrow v\}$

$\texttt{DoWrite}_t(l, v)$
Pre: $pc_t = \mathrm{doWrite}(l, v)$
Eff: $pc_t := \mathrm{writeResp}$
$wrSet_t := wrSet_t \oplus \{l \rightarrow v\}$

where

$$
\begin{aligned}
PCExternal &\;\widehat{=}\; \{\mathrm{notStarted}, \mathrm{ready}, \mathrm{commitResp}, \mathrm{writeResp}, \mathrm{committed}, \\
&\qquad \mathrm{aborted}\} \cup \{\mathrm{readResp}(v) \bullet v \in V\} \\
PC &\;\widehat{=}\; PCExternal \cup \{\mathrm{beginPending}, \mathrm{doCommit}, \mathrm{cancelPending}\} \\
&\qquad \cup \{\mathrm{doRead}(l) \bullet l \in L\} \cup \{\mathrm{doWrite}(l, v) \bullet l \in L, v \in V\} \\
maxIdx &\;\widehat{=}\; max(\mathrm{dom}(memories)) \\
latestMem &\;\widehat{=}\; memories(maxIdx) \\
readCons(mem, rdSet) &\;\widehat{=}\; rdSet \subseteq mem \\
validIdx(t, n) &\;\widehat{=}\; beginIdx_t \leq n \leq maxIdx \wedge readCons(memories(n), rdSet_t)
\end{aligned}
$$

Fig. 2. The state space and transition relation of TMS2 [6]

For TMS2 the shared state is a nonempty sequence of *memories*, which can be indexed by $n \in \mathbb{N}$. Each element of this sequence results from a successfully committing transaction. It specifies a possible *view* on the memory a subsequent transaction sees. Besides this sequence, the state consists only of variables local to a transaction $t \in T$. The state of each transaction t is defined over a program counter $pc_t \in PC$, a numerical identifier of the transaction $beginIdx_t \in \mathbb{N}$, a read set $rdSet_t$ and a write set $wrSet_t$. The latter two map locations $l \in L$ to values $v \in V$. The program counter values are further distinguished into values, in which the transaction is either not yet started or already finished ($PCExternal$) and the remaining program locations, in which it is active. The $beginIdx_t$ marks the memory with the smallest index, that transaction t may use when reading and writing. When other transactions finish while t is running, new views become available that t may use.

The relation $steps(TMS2)$ contains three transitions for each of the operations of a transaction (begin, read, write, commit, cancel). Each operation has a separate invoke and response transition that is *externally* visible. These capture the idea that each operation may take time and thus can be refined by multiple concrete steps. The response to $inv_t(\texttt{cancel})$ is always $resp_t(\texttt{abort})$; begin, read and write may also respond by aborting the transaction. In addition, TMS2 provides steps for each operation, which are prefixed by a "*do*", e.g., $DoCommitWriter_t$. These are *internal* transitions at which an operation takes effect and becomes visible to other transactions (begin and cancel have an empty effect, therefore they have no *do*-step). These are similar to linearization points in linearizable data structures [16]. However, here, they only refer to one of the read, write or commit operations, and not to the complete transaction.

An IOA like TMS2 has *runs* by iteratively executing steps. Formally, a *run* of A is a (finite or infinite) sequence $s_0 a_0 s_1 a_1 s_2 \ldots$ of alternating states and actions of A, such that $s_0 \in start(A)$ and $(s_i, a_i, s_{i+1}) \in steps(A)$ for all i. A trace of a run is its largest subsequence consisting only of external actions. The set $trace(A)$ consists of all traces that are a trace of a run of A. It defines the possible observable behavior of A.

An automaton C refines automaton A, when $ext(C) = ext(A)$. The refinement is correct (we write $C \leq A$), when $trace(C) \subseteq trace(A)$ holds. The refinement relation between FastLane (as C) and TMS2 (as A) is shown via a *forward simulation*.

Theorem 1. *A* forward simulation *from a concrete IOA C to an abstract IOA A is a relation $F \subseteq states(C) \times states(A)$ such that each of the following holds:*

Initialization:
$\forall\, s_C \in start(C) \bullet \exists\, s_A \in start(A) \bullet F(s_C, s_A)$

External step correspondence
$\forall\, s_C, s'_C \in states(C), s_A \in states(A), a \in ext(C) \bullet F(s_C, s_A) \wedge (s_C, a, s'_C) \in steps(C)$
$\Rightarrow \exists\, s'_A \in states(A) \bullet F(s'_C, s'_A) \wedge (s_A, a, s'_A) \in steps(A)$

Internal step correspondence
$\forall s_C, s'_C \in states(C), s_A \in states(A), a \in int(C) \bullet F(s_C, s_A) \wedge (s_C, a, s'_C) \in steps(C)$
$\Rightarrow F(s'_C, s_A) \vee \exists s'_A \in states(A), a' \in int(A) \bullet F(s'_C, s'_A) \wedge (s_A, a', s'_A) \in steps(A)$
The existence of a forward simulation implies $C \leq A$.

The initialization condition requires initial states that match via F in both IOAs. External spreeps have to correspond in both IOAs preserving F. Internal steps of C must match either one internal or no abstract step, again preserving the forward simulation F. Thus by induction, for all runs of C a corresponding run of A can be found with the same trace, implying refinement. We write $C \leq_{(F)} A$, when refinement of A to C is provable with forward simulation F. The forward simulation conditions given here are derived from those in Lynch and Vandrager [20] (adapted to several internal actions). The same proof obligations are also used in our opacity proof of a pessimistic STM in [8].

4 Proving FastLane Refinement of TMS2

In this section we prove that the FastLane algorithm of Fig. 1 is opaque, by providing a corresponding automaton *FastLane*, a forward simulation F and proving $C \leq_F TMS2$. That a combination with other algorithms is possible, is shown in the next section. The states and some example transitions of *FastLane* are shown in Fig. 3. The state variables directly correspond to the global and local variables of the algorithm. Since we want to distinguish one thread as master, it is necessary to have program counters pc_p for every thread p instead of for every transaction. $pc_p \in \{L1 \ldots L74\}$ gives the line number the thread will execute next. A thread that does not execute a transaction has $pc_p = L0$. A thread is in between executing reads and writes, has $pc_p = M0$ if is the master, $pc_p = H0$ otherwise. Since our setting involves processes, it is now necessary to manage a finite map $pidf : PID \nrightarrow T$ between threads and transactions. To have a simple scheme to get a new transaction, we assume, that transactions $T(0), T(1), \ldots$ can be enumerated. Starting a new transaction by invoking START (action inv_t(`TMBegin`) in Fig. 3) uses a global counter *tcntr*, that is incremented in this step, $T(tcntr)$ is the new transaction that is put into *pidf*.

We typically have one transition per line of code, plus additional steps labelled with the corresponding external action of TMS2 for invoke and return. All other steps are labelled with internal actions which are parameterized with the thread p executing it. They usually are named with helper or master routine executed, and the line number, $start2_p$ and $helperstart39_p$ are two examples. As an exception, those steps of the program that correspond to *do*-steps of TMS2 are labelled with the corresponding internal action of TMS2 (we have $act(TMS2) \subseteq act(C)$). This allows to already fix the internal step correspondence in the forward simulation of Theorem 1. Steps with actions from TMS2 require a corresponding abstract step, all others require none.

Some example transitions of the FastLane automaton are shown at the bottom of Fig. 3 definitions. The full transition relation can be viewed as part

State variables:
$mem : L \to V$, initially $mem = \lambda\, l \bullet 0$
$dirty : L \to V$, initially $dirty = \lambda\, l \bullet 0$
$masterID : PID$, initial master is arbitrary
$masterLock : PID \cup \{\bot\}$, initially $masterLock = \bot$
$addr_t : L, val_t : V, start_t : \mathbb{N}$, arbitrary initially
$cntr : \mathbb{N}$, initially $cntr = 0$
$pidf : PID \nrightarrow T$, initially $pidf = \varnothing$
$helpers : PID^*$, initially $helpers = \langle\,\rangle$
$tcntr : \mathbb{N}$, initially $tcntr = 0$
$pcf_p : PC$, initially $pcf_p = L0$
$rdsetf_t :: L \nrightarrow V, wrsetf_t : L \nrightarrow V$, initially $rdsetf_t = wrsetf_t = \varnothing$

Example transitions:

$inv_t(\texttt{TMBegin})$
Pre: $pcf_p = L0$
$p \notin \mathrm{dom}(pidf)$
$t = T(tcntr)$
Eff: $pcf_p := L2$
$pidf := pidf \cup \{(p, T(tcntr))\}$
$tcntr := tcntr + 1$

$\texttt{DoRead}_t(addr_t, maxIdx)$
Pre: $pcf_p = L18$
$pidf(p) = t$
Eff: $val_t := mem(addr_t)$
$pcf_p := L18b$

$start2_p$
Pre: $pcf_p = L2$
$pidf(p) = t$
Eff: $pcf_p :=$ **if** $masterID = p$ **then** $L3$ **else** $L9$

$resp_t(\texttt{TMRead}(val_t))$
Pre: $pcf_p = L18b$
$pidf(p) = t$
Eff: $pcf_p := M0$

$helperstart39_p$
Pre: $pcf_p = L39 \wedge pidf(p) = t$
Eff: $pcf_p := L40$
$start_t :=$ **if** $even(cntr)$ **then** $cntr$
else $cntr - 1$

Fig. 3. Definition of state and some example transitions as part of the transition relation of the FastLane mode.

of our proof project website [11]. The examples show that line 18 of the code is split in two steps. A first step that reads $mem(addr_t)$ into the local variable val_t, which implements the $\texttt{DoRead}_t(addr_t, maxIdx)$ step from TMS2 (the index $maxIdx$ fixes that the read of TMS2 will be from $lastMem$), and the returning step which implements $resp_t(\texttt{TMRead}(val_t))$.

For the steps that implement *do*-steps of TMS2 we choose

Action	Master Line	Helper Line
$\texttt{DoCommitReadOnly}_t(n)$	L27(if condition negative)	L56 (if condition positive)
$\texttt{DoCommitWriter}_t$	L28	L68
$\texttt{DoRead}_t(addr_t, maxIdx)$	L18a	L43 & L44
$\texttt{DoWrite}_t(l, v)$	L24	L53

The remaining steps of the FastLane mode are defined in a similar way. All of them have to be atomic steps of the behavior, and thus impossible to divide into more fine-grained steps.

To prove opacity we created a forward simulation, a representation of its specification is shown below:

$$MemR(s_C, s_A) \wedge \forall\, p \bullet FW(s_A, s_C, p)$$

Each element of the specification either serves the purpose of specifying the forward simulation or to narrow the scope of states that need to be considered. The $MemR(s_C, s_A)$ predicate only lets concrete and abstract states be matched, if the current memory and the write sets of the abstract state are consistent with the memory of the concrete state. The FW predicate contains the invariant and the remaining forward simulation. The predicates are described in more detail in the following:

MemR The predicate uses a threefold case distinction. If *cntr* is even (which means no writer is currently active), then $latestMem = mem$ must hold. If *cntr* is odd, there a two options possible:

- $masterID = masterLock$: Let $T(x)$ be the transaction currently running on the thread *masterLock*. The currently writing thread is the master thread thus all writes of its writeset have been executed. Thus $latestMem \oplus wrset(T(x))$.
- $masterID \neq masterLock$: Then the currently writing transaction is a helper transaction in a committing state. For every location one of two cases holds. If $dirty(l) \neq cntr$ then the helper transaction did not write to l thus $mem(l) = latestMem(l)$ must hold.
  If $dirty(l) = cntr$ then $mem(l)$ must already contain the value from the corresponding write set.

FW The first part of *FW* specifies the forward simulation. This is the mapping of program counters from the TMS2 automaton to the FastLane specification. It matches all internal "do" and external invoke or commit actions to the corresponding abstract actions.

The second part - the invariant - specifies multiple facts that need to hold for concrete or abstract states to be part of the simulation. For concrete states these facts need to hold for all reachable states, since otherwise the forward simulation can not be shown. The invariant excludes impossible states from consideration and so simplifies the proof.

For abstract states it is specified that in their initial state all write and read-sets are empty and all transactions are still either in a notStarted or beginPending program counter. For concrete states mainly the relation of the current program counter to certain variables is specified. A key point is the restriction on the *dirty* function which is used in the proof to relate the abstract and concrete memory. Additionally it is stated when *masterLock* is definitely possessed by some thread and when it is not. Similarly it is specified if *cntr* is certain to be even or odd with regards to program counters. Also certain nonsensical states are excluded, e.g. two threads having the same transaction.

5 Combining FastLane with Other Implementations

The Fastlane algorithm is optimized for the case, where a small number of threads is active, which usually implies that conflicts between transactions are

rare. When the number of active processes becomes greater than some threshold c, it is assumed, that the implementation switches to another implementation ([24] suggests e.g. [5]), that is better suited to handle large numbers of parallel transactions. To further increase efficiency, it is assumed that as long as only a single process is active, the execution switches to a purely sequential implementation where starting and finishing a transaction does nothing, and reading and writing directly accesses the main memory. We have modelled such a sequential implementation SEQ as an IO-Automaton as well. The implementation uses a single program counter, so by construction it is able to run at most one transaction t at any time. We have verified $SEQ \leq_{(F)} TMS2$ using a simple forward simulation F that maps the current memory *mem* used in *SEQ* to $lastMem \oplus wrset(t)$ in TMS2 while transaction t is running (otherwise directly to *lastMem*)[3].

The question we therefore have to solve in this section, is how to define and to verify a combined model *COMB* which switches between three modes: *SEQ*, FastLane and an (unknown) third implementation *IMPL* of *TMS*2. The automaton *COMB* will require additional state variables and transitions compared to the individual ones. The state of *COMB* must store the current mode, the number of currently active processes, and a mapping between processes and transactions. Note that a process can execute several transactions, while each transaction $t \in T$ can be executed by TMS2 only once.

To modularize the problem we give a solution in two steps. The first step gives generic criteria for combining two different implementations $C1 \leq_{(F1)} A$ and $C2 \leq_{(F2)} A$ into a new implementation $C \in switch(C1, C2)$, defined below, that switches between $C1$ and $C2$. More precisely, it specifies sufficient proof obligations under which $C \leq_{(F)} A$ for some F. Since we do not fix any specifics of C or A, the theorem we derive for this step is generally valid for all implementations of interfaces combining several ones into one.

The second step then instantiates the generic theorem to prove that our concrete combination *COMB* is in $switch(switch(SEQ, FastLane), IMPL)$.

Definition 2. *Given two automata $C1$ and $C2$ an automaton C is in $switch(C1, C2)$, if it satisfies the following criteria:*

- $ext(C) = ext(C1) = ext(C2)$ *and* $int(C_1) \cup int(C_2) \subseteq int(C)$.
- *All states $s_C \in states(C)$ have a boolean* mode*-component $s_C.mode$ to determine which algorithm is active.*
- *All states $s_C \in states(C)$ with $s_C.mode = true$ allow to extract via $s_C.s1 \in states(C1)$. Similarly, there is a selector $s_C.s2 \in states(C2)$, when $s_C.mode = false$.*
- *Initial states $s_C \in start(C)$ satisfy $s_C.mode = true \wedge s_C.s1 \in start(C1)$ (the first implementation starts).*
- *The transition relation of $step(C)$ can be split into three parts $step1, step2$, $step3$: $step1$ and $step2$ must correspond to steps of $C1$ and $C2$, $step3$ are new internal steps. Formally, it is required, that the following holds:*

[3] Details of this proof can be found in the corresponding KIV project [11].

$$step1(C)(s_C, a, s'_C) \Rightarrow step(C1)(s_C.s1, a, s'_C.s1) \wedge s_C.mode \wedge s'_C.mode,$$
$$step2(C)(s_C, a, s'_C) \Rightarrow step(C2)(s_C.s1, a, s'_C.s2) \wedge \neg\, s_C.mode \wedge \neg\, s'_C.mode,$$
$$step3(C)(s_C, a, s'_C) \Rightarrow a \in int(C).$$

The simplest automaton $C \in switch(C1, C2)$ has $states(C) = states(C1) \times states(C2) \times Boolean$, where $.s1$, $.s2$, and $.mode$ select the three components. Our more liberal definition allows to add extra state, and to avoid duplication of state. E.g., both *SEQ* and *FastLane* will use main memory, which we do not want to duplicate. Note that the implication allows for $step1(C)$ to have additional preconditions compared to $step(C1)$, or to have additional effects on state variables not selected by $s_C.s1$. The steps in $step3$ are new internal steps added to the automaton that manage switching between modes. To prove that C refines A we need a new forward simulation. It is clear that we need $F1$ when $s_C.mode = true$ and $F2$ when $s_C.mode = false$. However, this is typically not sufficient, as other new components of the state will need an extra invariant $Inv(s_C)$ to "glue" the parts together. Also, while $C1$ is running, critical state variables needed by $C2$ should not be changed, which gives another invariant $Inv1(s_C)$. In our example, running the sequential algorithm must keep *cntr* at an even value, *masterLock* free (i.e. $= \bot$), and preserve $dirty(l) \leq cntr$ for all $l \in L$. Therefore we define the following simulation relations $F0$ and F:

$$\begin{aligned} F0(s_C, s_A) \Leftrightarrow\ & \textbf{if } s_C.mode \textbf{ then } F1(s_A, s_C.s1) \wedge Inv1(s_C) \\ & \textbf{else } F2(s_A, s_C.s2) \wedge Inv2(s_C) \\ F(s_C, s_A) \Leftrightarrow\ & Inv(s_C) \wedge F0(s_C, s_A) \end{aligned}$$

We can then prove the following theorem:

Theorem 2. *Let $C1 \leq_{(F1)} A$, $C2 \leq_{(F2)} A$ and $C \in switch(C1, C2)$. Then $C \leq_{(F)} A$ with F defined as above holds, when the following proof obligations can be shown:*

- *All transitions preserve the invariant:*
  $Inv(s_C) \wedge step(C)(s_C, a, s'_C) \Rightarrow Inv(s'_C)$
- *All step2-transitions preserve Inv1:*
  $Inv(s_C) \wedge Inv1(s_C) \wedge step(C)(s_C, a, s'_C) \wedge step1(C)(s_C.2, a, s'_C.2) \Rightarrow Inv1(s'_C)$
- *All step1-transitions preserve Inv2:*
  $Inv(s_C) \wedge Inv2(s_C) \wedge step(C)(s_C, a, s'_C) \wedge step2(C)(s_C.2, a, s'_C.2) \Rightarrow Inv2(s'_C)$
- *Switching steps preserve F0:*
  $Inv(s_C) \wedge F0(s_C, s_A) \wedge step3(s_C, a, s'_C) \Rightarrow F0(s'_C, s_A)$

Proof: The proof obligations imply that F is a forward simulation. For steps in $step1$ F in the pre- and poststate both reduce to $F1 \wedge Inv1$ which is preserved since $C1 \leq_{(F1)} A$ and the third condition. Similarly for steps in $step2$. For switching steps, the first and last proof obligations directly show that $F0$ and Inv hold afterwards. □

We apply the theorem twice. First, we define $SFL \in switch(SEQ, FastLane)$, then $COMB \in switch(SFL, IMPL)$ parameterized with $IMPL$, where we just assume $IMPL \leq_{F3} TMS2$. For simplicity, both SFL and $COMB$ use the same states, they differ only in the available transitions. Each state s_C is a tuple of all the state variables of SEQ and $FastLane$ (so selectors $s_C.seq$ and $s_C.fl$ are definable), plus an arbitrary rest component for $IMPL$, such that a selector $s_C.impl \in states(IMPL)$ can be assumed. The tuple also includes $runmode \in \{seq, fl, impl\}$ to determine the running machine and a finite set $regset \in set(PID)$ of (registered) process identifiers. The processes running transactions (those in $\mathrm{dom}(pidf)$) are required to be a subset of this set. The initial states of both SFL and $COMB$ satisfy $s_C.seq \in start(SEQ) \wedge runmode = seq \wedge pidf = \varnothing \wedge regset = \varnothing$. SFL runs transitions of SEQ, when $s_C.runmode = seq$, otherwise steps of $FastLane$. Theorem 2 is instantiated by defining $s_C.mode := (runmode = seq)$ and by using their steps when defining $step1$ resp. $step2$.

As a first part of $step3$, both SFL and $COMB$ have transitions to register and unregister a process pid by adding or removing it from $s_C.regset$, leaving all other state unchanged. Unregistering pid requires $pid \notin \mathrm{dom}(pidf)$. These steps leave the running implementation $s_C.runmode$ unchanged. Steps that switch the running implementation form the second part of $step3$. They change $runmode$ to seq, when $card(regset) \leq 1$, to $FastLane$ when $card(regset)$ is below some constant c, and to $IMPL$ otherwise. These switching steps are enabled when no transaction is running, i.e. when $pidf$ is empty.

Finally, steps that start or leave transactions (of each machine) are restricted to modify $pidf$ suitably (all others leave them unchanged). Steps that start a new transition t (have action $inv_t(\texttt{TMBegin})$ of $TMS2$) must be invoked by some $pid \in regset$ that does not run a transition. They add the pair (pid, t) to $pidf$. An additional precondition ensures, that they are not enabled, when the system wants to switch to a new implementation. Dually, steps that finish a transaction (have action $resp_t(\texttt{abort})$ or $resp_t(\texttt{TMEnd})$) remove the corresponding pair again.

The additional invariant Inv for the SFL instance requires that processes running transactions are registered ($\mathrm{dom}(pidf) \subseteq regset$), and that transactions with a number $\geq tcntr$ are not in $pidf$ and still have empty read- and writesets. $Inv1$ stores properties of the FastLane state, that hold while the sequential algorithm is running: $cntr$ is even, $masterLock$ is free, all program counters pcf_p are $L0$, and $dirty(l) \leq cntr$ holds for all locations l. $Inv2$ is not used (i.e. $true$).

For the second instance $COMB$ we have to be careful that reusing state of $FastLane$ in $IMPL$ is possible (e.g. $masterLock$ or the $dirty$ flags might be shared with $IMPL$). Therefore predicates Inv, $Inv1$ and $Inv2$ as well as the effect of switching to and from $mode = impl$ on other state variables are left unspecified. This allows to either specify that state is not reused (e.g. $masterLock = free$ as part of $Inv2$) or to constrain switching (e.g. switching to fl may require a state with $masterLock = free$, or alternatively may reset $masterLock$). An instance must be provided for each concrete implementation $IMPL$, and we collect the necessary assumptions that must be proved. A full listing of all assumptions

can be found in the KIV proofs at [11], here we only give a few typical ones informally (recall that $IMPL \leq_{F3} TMS2$ and $SFL \leq_F TMS2$).

- Register and Unregister in $mode = impl$ preserve $F3$ and $Inv1$.
- Switching from $mode = impl$ may assume Inv, $Inv2$ and $F3$ before the step. After the step F and $Inv1$ must hold, together with the invariants Inv and $Inv1/Inv2$ of SFL (as specified above), when the new mode is seq/fl.
- Steps of $IMPL$ as well as steps that switch to or from $mode = impl$ must leave $regset$ unchanged.
- Steps of $IMPL$ must change $pidf$ correctly (similar to $FastLane$).

6 Conclusion

In this paper, we presented the results of a case study on proving opacity of STMs, specifically FastLane. The case study required the development of proof techniques for *combined* STMs, where the combination concerned the switching between modes. This general proof technique has been shown to be sound, and has been instantiated for FastLane. All proofs have been fully mechanized with the theorem prover KIV.

Related Work. A number of other approaches for proving opacity of STMs have been proposed. These can be classified as either *model checking* or *deductive verification* techniques. In model checking, there has been research by Guerraoui et al. [12,13] using a reduction theorem; other approaches use model checkers like SPIN or TLC [4,21].

In deductive verification, the most frequently used approach is to show a given STM to be a refinement of TMS2 [2,6,8,9,17]. The only instance known to us, where a similar type of "combined" STM was considered is work by Armstrong and Dongol [1]. They show refinement to hold between a concrete TM consisting of a hardware and a software TM and TMS2. The main difference to the scenario considered here is that there both TMs run in parallel. Thus – in contrast to our setting – interference between TMs needs to be modelled as well which is done with the help of so-called *interference* automata. Then, a forward simulation is shown to exist between each of the automata and TMS2. Under some constraints on the interferences, these forward simulations can be combined to give a simulation between the complete TM and TMS2. In flavour, this is similiar to our approach, but we use switching for combining TMs, not parallel composition.

References

1. Armstrong, A., Dongol, B.: Modularising opacity verification for hybrid transactional memory. In: Bouajjani and Silva [3], pp. 33–49. https://doi.org/10.1007/978-3-319-60225-7_3

2. Armstrong, A., Dongol, B., Doherty, S.: Proving opacity via linearizability: a sound and complete method. In: Bouajjani and Silva [3], pp. 50–66. https://doi.org/10.1007/978-3-319-60225-7_4
3. Bouajjani, A., Silva, A. (eds.): FORTE 2017. LNCS, vol. 10321. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60225-7
4. Cohen, A., O'Leary, J.W., Pnueli, A., Tuttle, M.R., Zuck, L.D.: Verifying correctness of transactional memories. In: Formal Methods in Computer Aided Design, 2007, FMCAD 2007, pp. 37–44. IEEE (2007)
5. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: streamlining STM by abolishing ownership records. In: Govindarajan, R., Padua, D.A., Hall, M.W. (eds.) PPoPP, pp. 67–78. ACM (2010). http://doi.acm.org/10.1145/1693453.1693464
6. Derrick, J., Doherty, S., Dongol, B., Schellhorn, G., Travkin, O., Wehrheim, H.: Mechanized proofs of opacity: a comparison of two techniques. Formal Aspects Comput. (2017). https://doi.org/10.1007/s00165-017-0433-3
7. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006). https://doi.org/10.1007/11864219_14
8. Doherty, S., Dongol, B., Derrick, J., Schellhorn, G., Wehrheim, H.: Proving opacity of a pessimistic STM. In: Fatourou, P., Jiménez, E., Pedone, F. (eds.) OPODIS. LIPIcs, vol. 70, pp. 35:1–35:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016). https://doi.org/10.4230/LIPIcs.OPODIS.2016.35
9. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Towards formally specifying and verifying transactional memory. Formal Aspects Comput. **25**(5), 769–799 (2013). https://doi.org/10.1007/s00165-012-0225-8
10. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV - overview and verify this competition. Softw. Tools Technol. Transf. **17**, 677–694 (2014)
11. KIV proofs for FastLane (2018). http://www.informatik.uni-augsburg.de/swt/projects/FastLane.html
12. Guerraoui, R., Henzinger, T.A., Singh, V.: Completeness and nondeterminism in model checking transactional memories. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 21–35. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85361-9_6
13. Guerraoui, R., Henzinger, T.A., Singh, V.: Model checking transactional memories. Distrib. Comput. **22**(3), 129 (2010)
14. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Chatterjee, S., Scott, M.L. (eds.) PPOPP, pp. 175–184. ACM (2008). http://doi.acm.org/10.1145/1345206.1345233
15. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC, pp. 92–101. ACM (2003)
16. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990). https://doi.org/10.1145/78969.78972
17. Lesani, M., Luchangco, V., Moir, M.: A framework for formally verifying software transactional memory algorithms. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 516–530. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32940-1_36
18. Lesani, M., Luchangco, V., Moir, M.: Putting opacity in its place. In: Workshop on the Theory of Transactional Memory (2012)
19. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: Schneider, F.B. (ed.) PODC, pp. 137–151. ACM (1987). http://doi.acm.org/10.1145/41840.41852

20. Lynch, N.A., Vaandrager, F.W.: Forward and backward simulations: I Untimed Systems. Inf. Comput. **121**(2), 214–233 (1995). https://doi.org/10.1006/inco.1995.1134
21. O'Leary, J., Saha, B., Tuttle, M.R.: Model checking transactional memory with Spin. In: 29th IEEE International Conference on Distributed Computing Systems, 2009, ICDCS 2009, pp. 335–342. IEEE (2009)
22. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979). http://doi.acm.org/10.1145/322154.322158
23. Shavit, N., Touitou, D.: Software transactional memory. Distrib. Comput. **10**(2), 99–116 (1997). https://doi.org/10.1007/s004460050028
24. Wamhoff, J., Fetzer, C., Felber, P., Rivière, E., Muller, G.: FastLane: improving performance of software transactional memory for low thread counts. In: Nicolau, A., Shen, X., Amarasinghe, S.P., Vuduc, R.W. (eds.) PPoPP, pp. 113–122. ACM (2013). http://doi.acm.org/10.1145/2442516.2442528

Program Analysis

Monte Carlo Tree Search for Finding Costly Paths in Programs

Kasper Luckow[1], Corina S. Păsăreanu[1,2(✉)], and Willem Visser[3]

[1] Carnegie Mellon University SV, Mountain View, CA, USA
ksluckow@gmail.com, corina.pasareanu@west.cmu.edu
[2] NASA Ames Research Center, Mountain View, CA, USA
[3] Stellenbosch University, Stellenbosch, South Africa
wvisser@cs.sun.ac.za

Abstract. We describe a heuristic analysis technique for finding costly paths in programs, where the cost refers to the execution time or memory consumed by the program. The analysis can support various software engineering tasks, such as finding vulnerabilities related to denial-of-service attacks, guiding compiler optimizations or finding performance bottlenecks in software. The analysis performs sampling over symbolic program paths, which are computed with a symbolic execution over the program, and uses Monte Carlo Tree Search (MCTS) to guide the search for costly paths. We implemented the proposed method in Symbolic PathFinder and we evaluated it on Java programs. Our experiments show the promise of the technique for finding performance bottlenecks in software.

1 Introduction

In this paper, we investigate the application of Monte Carlo Tree Search (MCTS) for the analysis of space-time consumption in programs. MCTS is a search heuristic that uses random sampling to iteratively expand a search tree with the goal of finding the optimal decisions in a given large problem domain. MCTS has found success in solving difficult Artificial Intelligence problems, notably in computer Go [12], which has been regarded as one of the most challenging games for AI. Despite its success, there has been little research reported on the use of MCTS for software analysis, although there is an obvious fit, since often software analysis artifacts can be expressed as annotated trees or graphs (e.g. abstract syntax trees, symbolic execution trees, call trees or graphs etc.) which can be naturally explored with MCTS.

In our work, we use MCTS to find *deep* or *costly* paths in programs, where the cost refers to execution time or memory consumed along a path. Our work is motivated by a DARPA project which addresses the detection of software worst-case vulnerabilities related to the space-time usage of software systems. An adversary can exploit these vulnerabilities by generating inputs that induce expensive space-time utilization, thereby denying service to begin users

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 123–138, 2018.
https://doi.org/10.1007/978-3-319-92970-5_8

or disabling the system. Finding paths with large space-time cost can also have many other applications, such as enabling compiler optimizations or finding and fixing performance bottlenecks. Our technique generalizes to finding paths that maximize other costs as well, such as input/output usage or power consumption along a path (provided a suitable hardware model). Furthermore, our technique can be applied, with small modifications, to other types of software analysis, e.g. to maximize coverage of statements and assertions.

Our analysis uses Monte Carlo sampling over program paths, assuming at first a uniform distribution over the choices at the conditions in the code, i.e. both *true* and *false* branches are assumed to be equally likely. The analysis then gradually builds a search tree that dictates how to change the probabilities of taking the branching conditions, to focus the search on "costly" paths in the program, where the cost records the execution time or memory consumed along the path. The output of the algorithm is a program path that likely leads to the highest cost in the program.

Instead of doing pure Monte Carlo sampling over the program, which may require a prohibitively large amount of samples, we chose to perform sampling over *symbolic program paths*, which are computed with a symbolic execution of the program. Each symbolic path represents multiple concrete paths all following the same control flow in the program. Thus we only need to sample each program path once and we can also *prune* the already sampled paths, to speed-up the convergence of the analysis to the optimal results. Our MCTS approach (with pruning) always converges to the results of an exact analysis but it is often much faster, as we show in the experiments.

We have implemented our technique using the Symbolic PathFinder (SPF) symbolic execution tool [9]. We evaluate it on Java implementations of classical algorithms, comparing it with: (exact) symbolic execution, Monte Carlo sampling, and a Reinforcement Learning approach adapted from [7]. In addition, we evaluate MCTS on complex systems from DARPA STAC engagements and show the merits of the approach compared to exact analysis.

2 Background

Monte Carlo tree search (MCTS) is a heuristic search algorithm for certain decision processes, most notably those used in game play. The algorithm focuses on the analysis of the most promising moves (in a game), expanding a search tree based on random sampling of the search space [1]. MCTS can generally be applied to any application domain that can be modeled as a tree. MCTS is *asymmetric* in the sense that it favors exploring promising (and seemingly important) sub-trees while never assigning a zero probability to any choice (governed by criteria discussed below).

Conceptually, MCTS consists of four steps which are repeated until some stopping criteria is met (number of iterations, quality of result, etc.).

1. *Selection:* start from the root node, R, in the search tree and select successive child nodes according to the *tree policy* down to an *expandable* node, L, i.e. a node that has unvisited children. L can also be a terminal node (e.g., win/loss for a game) in which case MCTS continues with step 4.
2. *Expansion:* for the expandable node, L, add a node, C, to the search tree from the available decisions at L.
3. *Simulation:* perform a random playout from node C governed by the *simulation policy* until a terminal node is reached. The terminal node yields an outcome, *reward*, based on the *reward function*.
4. *Backpropagation:* update the information stored in the nodes on the path from C to R with the *reward* computed. This might also include updating visit counts etc.

The specifics of the reward function depend on the application and the type of analysis being performed.

The MCTS-guided symbolic analysis we propose is based on the Upper Confidence Bound applied to Trees (UCT); the algorithms presented in this paper are adapted versions of the general UCT MCTS algorithm described in [1]. The UCT MCTS algorithm adapts the upper confidence bound (UCB1) algorithm from the multi-armed bandit literature to the tree setting and builds the search tree by expanding one node at a time prioritized by the UCB criteria. It guarantees an optimal balance of *exploration* and *exploitation*—that is, the trade-off between exploiting actions that are known to be good and exploring actions with poor value estimates. The algorithm directs sampling of paths to adaptively focus the search towards more promising areas of the search space. Heuristic approaches, as used in previous studies, do not have this property.

3 MCTS with Symbolic Execution

We describe a symbolic execution approach that uses MCTS to guide the search for the program paths that have the highest time or memory cost (i.e. reward). Symbolic execution is a systematic program analysis technique which executes programs on symbolic inputs instead of concrete inputs. For each explored path, the analysis maintains a *Path Condition* (PC), which is a conjunction of constraints over the symbolic inputs that characterize all the concrete inputs that follow the path. The symbolic paths form a symbolic execution tree that characterizes all the concrete program paths, up to some user-specified bound.

At a high level, our analysis works by gradually building and sampling the symbolic execution tree of the program. In our setting, a symbolic execution tree only encodes the *decisions* at conditional statements and we disregard the other states (e.g., assignments, method invocations, and returns). A decision is introduced whenever a conditional statement (on condition c) is symbolically executed in the program. The evaluation of the statement introduces two new transitions in the tree: The first one leads to the execution of the "then" block in the code and the path condition is updated as $PC_{then} = c \wedge PC$. The second leads to the execution of the "else" block and the path condition is updated with

$PC_{else} = \neg c \wedge PC$. If the path condition for a branch is not satisfiable, which is checked with an off-the-shelf solver, that branch is not explored.

We denote a symbolic execution tree augmented with information for MCTS as the tuple $T = \langle S, s_0, \rightarrow, S_t \rangle$, where S is the set of nodes (or states), $s_0 \in S$ is the initial state, $\rightarrow \subseteq S \times S$ is the transition relation, and $S_t \subseteq S$ is the set of decisions in the search tree maintained by the MCTS algorithm. Let $children(s)$ and $parent(s)$ denote the children nodes and parent node of state $s \in S$, respectively.

For a decision node, $s \in S_t$, $N(s)$ denotes how many times the decision has been visited, and $Q(s)$ denotes the cumulative reward of all the samples that have passed through this decision (the precise computation is defined below). S_t, $N(\cdot)$ and $Q(\cdot)$ are global to all algorithms in the following.

We also define a reward function, $reward(s)$, that for a terminal state $s \in S$ returns a value that quantifies the outcome of the sample. In our setting, $reward(s)$ denotes time or memory cost for a path. For our experiments, we use the number of nodes along the path in the tree as a proxy for timing measurement, and the number of bytes allocated along the path for memory measurement. Other reward functions can be defined in a straightforward way.

Algorithm 1 shows the overall approach.

Algorithm 1. Overall algorithm for MCTS-guided symbolic execution

```
1:  function MCTSGuidedSymExe(s_0, C, n) ▷ Where s_0 - initial state of the program, C - MCTS bias, n - budget (samples)
2:      Δ_max ← −1
3:      S_t ← {s_0}
4:      for i = 1 to n do
5:          s_f ← TreePolicy(s_0, C)                          ▷ Steps 1 and 2
6:          Δ ← SymbolicSimulation(s_f)                       ▷ Step 3
7:          Backpropagation(s_f, Δ)                           ▷ Step 4
8:          Δ_max ← max(Δ, Δ_max)
9:      end for
10:     return Δ_max
11: end function
```

The algorithm takes as input an initial state, s_0, a bias parameter, C (described later), and a sampling budget, n. Note that in practice, we use (possibly composite) termination criteria, e.g., stopping the analysis based on time, bounds on expected reward, and/or coverage metrics. It returns the maximum reward, $\Delta_{\max}$, observed from the n samples. In practice extra information is returned, e.g., the path condition of the path with reward $\Delta_{\max}$; solving it provides concrete test inputs that will exercise the same (costly) path.

The algorithm maintains a set S_t of nodes in the search tree (i.e. the nodes in the symbolic execution tree that were *expanded* by MCTS). Initially the S_t set contains only the initial state, s_0. The algorithm then proceeds to perform the four MCTS steps n times. Steps 1 (selection) and 2 (expansion) are performed by procedure TreePolicy shown in Algorithm 2, while steps 3 (simulation) and 4 (backpropagation) are performed by SymbolicSimulation and Backpropagation, respectively, as explained later in this section.

TreePolicy performs a symbolic execution where each decision that has only children that are already part of the search tree, will be resolved using

Algorithm 2. Algorithm for selecting a new leaf to expand in the search tree

```
1:  function TreePolicy(s, C)                ▷ Where s - symbolic state, C - MCTS bias
2:      while s is non-terminal do
3:          if EligibleChildren(s) − S_t ≠ ∅ then
4:              return Expand(s)
5:          else
6:              s_c ← BestChild(s, C)
7:              s ← s_c
8:          end if
9:      end while
10:     return s
11: end function
```

procedure BestChild (Algorithm 3), which selects the child with the highest UCT value.

The UCT formula for balancing exploitation and exploration when selecting the next node s' from node$s$ is determined by:

$$UCT(s, s') = \frac{Q(s')}{N(s')} + C\sqrt{\frac{2 \ln N(s)}{N(s')}}$$

Here $C > 0$ is the *bias parameter*, a constant that controls how much exploration to perform. For rewards in $[0; 1]$, $C = \sqrt{2}$ satisfies the Hoeffding inequality [4]. For rewards outside this range other values of C may yield better results. The first component of the UCT formula above corresponds to *exploitation*; it is high for moves with high average win ratio. The second component corresponds to *exploration*; it is high for moves with few simulations. At each *selection* step, the child state with the largest UCT value is picked. Note $N(s) = \sum_{s' \in children(s)} N(s')$.

Algorithm 3. Computing the best child of a search tree node

```
1: function BestChild(s, C) ▷ Where s - symbolic state s.t. s ∈ S_t, children(s) ∈ S_t, C - MCTS bias
2:     return argmax_{s' ∈ children(s)} Q(s')/N(s') + C·sqrt(2 ln N(s) / N(s'))
3: end function
```

For now, assume that EligibleChildren(s) is equivalent to $children(s)$ (we will see it has a different meaning when pruning is enabled). Symbolic execution proceeds from this state. The procedure returns when a decision in the search tree is explored such that one (or more) children are *not* part of the search tree. In this case, successive node selection (step 1) ends, and MCTS proceeds with expansion (step 2) encapsulated by procedure Expand (Algorithm 4): from the set of unexpanded children, MCTS selects uniformly at random a child node to be added to the search tree. This node is then returned and marks the end of step 2. We denote this as the new *frontier node*, s_f, in the search tree.

The frontier node is used for performing a playout using symbolic simulation (procedure SimulationPolicy, Algorithm 5): successor nodes are selected at

Algorithm 4. Algorithm for expanding a new leaf in the search tree

```
1: function EXPAND(s)                                   ▷ Where s - symbolic state
2:     choose s' ∈ ELIGIBLECHILDREN(s) − S_t uniformly at random
3:     N(s') ← 0
4:     Q(s') ← 0
5:     S_t ← S_t ∪ s'
6:     return s'
7: end function
```

random until the path terminates. At this point, the reward function is applied to the end state and the reward is returned to the overall MCTS algorithm, which is the end of step 3.

Algorithm 5. Algorithm for making a random playout

```
1: function SYMBOLICSIMULATION(s)                       ▷ Where s - symbolic state
2:     while s is non-terminal do
3:         choose s' ∈ ELIGIBLECHILDREN(s) uniformly at random
4:         s ← s'
5:     end while
6:     return reward(s)
7: end function
```

Backpropagation (see Algorithm 6) is the final step in MCTS-guided sampling and iteratively updates the total reward with Δ and increments the visit count for all states starting at s_f and ending in the root node, s_0.

Algorithm 6. Algorithm for backpropagating reward

```
1: function BACKPROPAGATION(s, Δ)                       ▷ Where s - symbolic state, Δ - reward
2:     while s is not null do
3:         N(s) ← N(s) + 1
4:         Q(s) ← Q(s) + Δ
5:         s ← parent(s)
6:     end while
7: end function
```

Pruning. Our sampling works over symbolic paths, representing multiple concrete paths that follow the same control flow in the program. Thus, it is not necessary to sample along the same path multiple times, since repeated sampling can not bring new information to the analysis. We can therefore *prune* the explored paths, reducing the search space and accelerating the convergence of the analysis, since pruning enforces that paths cannot be re-sampled, thus always providing new information when reinforcing choices.

Pruning is enabled by keeping a boolean flag for each state, $p(s)$, marking whether s has been pruned from the search. When no pruning is used, procedure ELIGIBLECHILDREN(S) simply returns $children(s)$. When pruning is used, the definition of ELIGIBLECHILDREN is restricted to *only* include children states $s' \in children(s)$ s.t. $p(s') = false$.

```
1  void target(int a, boolean c) {
2    int i = 0;
3    if (c)
4      while (i < a) i++;
5    else
6      while (i < 10 * a) i++;
7  }
```

Listing 1. Example.

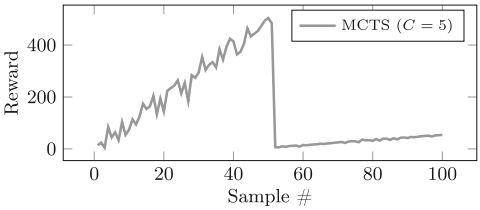


Fig. 1. Results: samples vs. reward

Initially, $p(s) = false$ for all states. When a path $s_0, s_1, \ldots, s_n$ terminates in TreePolicy (line 10) or SymbolicSimulation (line 6), the terminal node is updated with $p(s_n) = true$. This information is backpropagated iteratively for s_k where $k = n-1, n-2, \ldots, 0$ either until s_0 is marked (in which case the analysis terminates, because all paths have been explored), or when there is a $p(s'_k) = false$ such that $s_{k'} \in children(s_k)$ (i.e. all children must be marked as pruned before marking a parent as pruned).

Pruning also requires a modification to the expansion step (Algorithm 4): When a node s' is expanded from s, it is initialized with the visit counts and rewards obtained from playouts from that subtree of s. This is required since—if initialized to zero as in the non-pruning case—the qualities of the choices of s' would be incorrect because pruned paths cannot be re-sampled.

Since MCTS with pruning always explores new paths, and since we assume a finite number of paths, it follows that the search converges to an exhaustive analysis and that the path with the maximum reward (according to the reward function) will be sampled eventually.

Proposition 1 (Termination). *If pruning is enabled, then the symbolic path with the maximum reward is guaranteed to be sampled within n iterations of the MCTS algorithm, where n denotes the total number of symbolic paths.*

4 Example

We illustrate our approach on the example in Listing 1. This example shows a method that takes two symbolic inputs: one boolean input, c, that directs which of the two loops will be executed and an integer, $a \in [1, 50]$, that determines how many times each loop will execute. Note that the second loop will run 10 times as long as the first; it contains the longest path (maximum reward). Figure 2 shows the search tree that is built for the first four iterations by the analysis using a reward function based on depth of paths and bias $C = 5$. Note that we do not present unsatisfiable paths, e.g., the path condition $0 \geq a \cdot 10$ is not satisfiable given the constraints on a.

Initially, the search tree has only one node (the root). During the selection step (Algorithm 2) the initial state of the search tree has eligible children

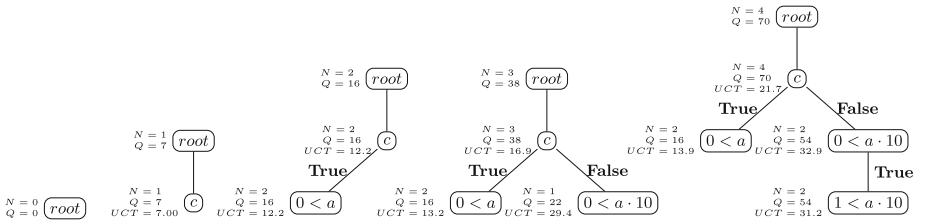


Fig. 2. Search trees obtained in the first four iterations.

(in fact just one[1]), and it is expanded (Algorithm 4) creating a node labeled with condition c. A simulation is started (Algorithm 5) and a random choice for c is made which can pick either the first or the second loop; let us assume the first loop is picked (*true* branch) and a simulation is made until the end, yielding the reward 7. This corresponds to 5 loop iterations before the simulation (randomly) decides to follow the *false* branch of the loop condition. The program terminates and the reward is propagated back up the tree (Algorithm 6). In the next iteration, one of c's children is expanded. This is also done randomly (Algorithm 4); assume that the child corresponding to the *true* branch is picked. Assume we use pruning, hence this node will be initialized with the rewards and visit counts already obtained for that sub-tree (i.e. the single run from before with reward 7). Now, from the playout, this path cannot be chosen again, but instead the loop is iterated seven times yielding reward 9. It is similarly propagated back and the cumulative reward for the nodes in the search tree will therefore be 16.

In the next iteration, MCTS is forced to expand the *false* branch for condition c (it is the only unexpanded child). Since this loop will iterate ten times more than the other loop, we also can expect to obtain rewards that are ten times higher. Let us assume that the simulation from this node yields the reward 22. Similarly, this value is propagated back, but note how it influences the UCT calculation for each node: for the next iteration of MCTS, the tree policy will decide to take the *false* branch of condition c again, because the UCT value is higher (Algorithm 3). A new node is expanded and initialized with the rewards and visit counts already obtained for the sub-tree due to pruning. Pruning enforces that the loop cannot be iterated the same number of times as before, but let us assume that a new path yields the reward 32. This value is propagated back thereby updating the UCT values of the nodes. For the next iteration (not shown here), MCTS will again pick this same branch, because the UCT is higher than for the *true* branch. Note that the UCT value is increasing at a much greater rate for the *false* branch (with the deep loop) than for the *true* branch due to the exploitation term: effectively MCTS will keep exploring the deep loop exhaustively before continuing with the smaller loop.

Our analysis tool allows the visualization of various statistics during execution and Fig. 1 shows the reward reached across the samples, and illustrates how

[1] For technical reasons, the root always has one child, representing the first condition.

the analysis favors the second loop at first (allowing the discovery of the longest program path). Only at the end does it try the first loop (the much lower part of the graph).

5 Implementation and Evaluation

We evaluated the presented approach (denoted MCTS) by a comparison with: traditional, exhaustive symbolic execution (Exact), standard Monte Carlo sampling (MC)—which proceeds by randomly selecting a choice at any decision—and another Reinforcement Learning (RL) method introduced in [7] for sampling symbolic paths. The RL approach has been modified to reinforce choices on decisions similar to the MCTS algorithm presented here. At a high level, it works by sampling paths according to probabilities computed for choices (initially uniform). Similar to MCTS, rewards are backpropagated and later used for reinforcing (i.e. updating the probabilities) the choices that seemed promising for maximizing the reward. The RL approach has three parameters: number of samples, L, made with current probabilities before the reinforcement step; a history parameter, h, that controls how much the probabilities are updated during the reinforcement step according to the rewards obtained from the L samples; and ϵ, which is a greediness parameter that adjusts the probability assigned during the reinforcement step to the best choice at a decision (i.e. the choice at a decision with the best game-theoretic value) from the L samples.

The major differences are that the RL approach is entirely probabilistic and uses two phases (sampling and reinforcing choices). In contrast, MCTS uses the deterministic *selection* phase after each sample to enforce sampling of the currently best sub-tree (according to the UCT value). To cope with potentially infinite paths arising from, e.g., loops, we perform a bounded exploration (with $k = 1000$ decisions) of the sampled paths to ensure termination. Precision of the analysis can be improved if k is exceeded by an iterative approach where k is gradually increased. For all our experiments, this limit was never reached.

For evaluating MCTS, RL and MC, we developed a framework based on Symbolic PathFinder (SPF) [9], called Canopy that enables experimenting with sampling-based symbolic analyses. Canopy is open-source and available at https://github.com/isstac/canopy which also contains a distribution for reproducing the results and an appendix with the full results tables. Canopy leverages two optimizations: incremental constraint solving and constraints satisfiability caching. The optimizations improve analysis times by a factor 2–5 with constraints caching dominating the improvements (full results are available in the appendix).

We used an application server running Ubuntu 16.04 and equipped with an Intel Xeon E5-2680@2.70 GHz and 64 GB of RAM. We use a reward function that counts the number of symbolic decisions along a path (i.e. the depth). Later, we also show experiments with a reward function that uses memory allocations along paths.

For the sampling-based approaches, we use a budget of 2,000 samples within which we record the results. Also, for all evaluations except the scalability evaluation, we made 50 runs with different seeds for the random number generators. We do this to demonstrate their general performance. For the scalability evaluation, we performed 25 runs for each data point.

Case Studies: We use two sets of case studies: *(1)* classic algorithms with well understood bounds to give the reader a better intuition of how MCTS performs; *(2)* complex applications (client-server, databases, native libraries, etc.), provided by DARPA as part of a project called STAC. *(1)* includes Quicksort, Merge Sort, Traveling Salesman Problem, Insertion Sort, and search operation in a Binary Search Tree. *(2)* includes *LawDB*, a network service hosting personnel data; *Gabfeed*, a web forum platform; *Text Crunchr*, a program for text encryption and analysis; *Airplan Shifter*, a web app for analyzing flight routes; *WithMi*, a P2P text chat and file transfer program; *Find Entry*, a program for managing entries in a data structure; and *Passwd Checker*, a password checking component. All examples have been parameterized by an input size (shown in parenthesis after their name) representing, e.g., the length of the list to be sorted for Quicksort, the size of a tree on which an operation is performed etc. Except for the scalability evaluation, we use the largest input size where the Exact analysis is still tractable within one hour of analysis time.

Evaluating the Effect of Pruning: We evaluated pruning vs. no pruning using MCTS $C = 5$—we will later show empirically that this a good overall bias parameter. Table 1 shows the results.

Column *Max* is the maximum reward observed over the 50 runs; column *Max #* is how many of the runs found the maximum observed reward; and column *Max #1run* is how many maximum rewards were found in a single run (i.e. 2,000 samples) on average.

When pruning is not used, the number of unique paths explored is significantly reduced; pruning, by construction, enforces exploration of new behaviors for each sample and guarantees progress. In terms of finding *one* path with the true maximum reward, pruning does not have a significant effect. However, pruning helps finding *multiple* paths with maximum reward. In particular, for BST Search, MCTS with pruning explores—in a single run of 2,000 samples—the two paths with maximum reward (i.e. the same as the exact analysis) out of over seven million paths.

Comparison of MCTS with MC, RL, and Exact: We evaluated MCTS and RL on several different configurations since the literature is not clear on their influence: MCTS with $C = \sqrt{2}, 5, 10, 20, 50, 100$; RL with the nine configurations corresponding to permutations of L, h and ϵ with high and low values.

We report the configuration for MCTS and RL that yielded the best overall results. The results table for all configurations (including raw data) is available at https://github.com/isstac/canopy.

Table 1. Pruning vs no pruning using MCTS $C = 5$.

Example	Paths			Max			Max #		Max #1run		
	Prun	No prun	Exact	Prun	No prun	Exact	Prun	No prun	Prun	No prun	Exact
BST Search(8)	2,000.00	893.40	7,087,261	82	79	82	2	1	2.00	2.00	2
Insertion Sort(10)	2,000.00	642.48	3,628,800	47	47	47	21	10	12.67	2.00	512
Merge Sort(10)	2,000.00	1,202.12	3,628,800	32	32	32	33	32	177.48	170.28	35,840
Quicksort(9)	2,000.00	1,642.72	1,081,621	53	53	54	2	6	32.00	4.33	160
TSP(5)	2,000.00	858.24	15,018	66	66	66	40	29	129.73	49.07	808
Airplan Shifter(10)	2,000.00	1,986.82	3,628,800	28	28	28	50	50	140.14	136.06	215,040
Find Entry(6)	2,000.00	873.30	660,665	86	80	86	2	1	2.00	2.00	2
Gabfeed(7)	2,000.00	1,262.56	973,049	62	62	62	6	4	1.17	1.25	64
LawDB(7)	2,000.00	1,069.04	545,835	48	48	48	3	15	40.00	2.73	240
Ngram(14)	2,000.00	431.82	7,174,453	42	41	44	2	3	2.00	2.00	2
Passwd Checker(4)	2,000.00	571.04	1,082,401	131	131	131	3	1	2.00	2.00	2
Text Crunchr(7)	2,000.00	68.30	1,149,877	65	59	65	2	1	12.00	1.00	64

Table 2 summarizes the results. Column *Max Rew.* presents the mean, μ, and standard deviation, σ, of the maximum rewards found from the 50 runs. They characterize the stability of the analysis in terms of finding maximum rewards despite different seeds. Column *Mean Rew.* presents the mean and standard deviation of all the paths covered in the runs and thus characterizes in general how "good" the technique is in exploring promising paths. Columns *Max*, *Max #* and *Max #1run* are the same as in Table 1. Columns *Max sample* and *Max time* show at which sample and time the path with the best reward from column *Max* was found on average.

MCTS is superior on a number of parameters: MCTS generally obtains—on average—a greater maximum reward than RL and MC. On every example, it also finds a path with equal or greater reward than RL and MC and in addition it tends to find multiple such paths when present. Compared to the Exact analysis, MCTS is also capable of finding the path with highest reward much faster. For example, for BST Search, MCTS finds one of two longest paths (out of over seven million paths) in less than 14 s, while the Exact analysis finds it after 44 min. From the experiments, a small bias parameter of $C = 5$ or $C = 10$ performs well overall for the sample budget. In contrast, a very high bias parameter yields poor results. This is expected since a high bias parameter will favor the exploration term thereby reducing MCTS to MC in the limit.

The plots in Figs. 3 and 4 show how MCTS ($C = 5$) and MC perform during 2,000 samples for Quicksort(9). MCTS is clearly better: it keeps exploring promising sub-trees finding the longest path (reward 53) in 488 samples while MC only finds a path with max reward 46 in 1,772 samples.

Table 2. Results for exact analysis, MCTS, and MC with path pruning. Number following the example name denotes the *input size* configuration. MCTS and RL configurations are denoted in parenthesis with the format MCTS(C) and RL(L;h;ϵ).

Example	Analysis	Max Rew. μ	Max Rew. σ	Mean Rew. μ	Mean Rew. σ	Max	Max #	Max #1run	Max sample	Max time [s]	Paths	Time [s]
BST Search(8)	Exact	82.00	—	38.95	6.12	82	—	2.00	5,774,480.00	2,617.00	7,087,261	3,203.00
	MC	40.20	1.20	29.24	2.57	43	2	1.00	1,416.50	13.50	2,000	17.18
	MCTS(5)	67.36	6.36	45.43	10.84	82	2	2.00	1,559.50	15.00	2,000	18.08
	RL(10;0.5;0.5)	46.02	3.39	34.30	3.37	55	1	2.00	998.00	8.00	2,000	17.30
Insertion Sort(10)	Exact	47.00	—	31.57	5.06	47	—	512.00	1.00	0.00	3,628,800	1,586.00
	MC	32.30	1.69	18.21	3.13	37	1	1.00	28.00	6.28	2,000	14.16
	MCTS(5)	45.96	1.05	32.47	7.72	47	21	12.67	1,089.67	8.44	2,000	16.94
	RL(100;0.1;0.1)	39.10	2.49	27.69	5.08	47	1	16.00	1,479.00	10.74	2,000	15.72
Merge Sort(10)	Exact	32.00	—	26.75	2.36	32	—	35,840.00	5.00	0.00	3,628,800	1,824.00
	MC	30.44	0.73	19.67	4.08	32	3	1.00	1,086.00	8.33	2,000	16.10
	MCTS(10)	31.70	0.46	21.31	4.48	32	35	2.23	1,493.77	11.71	2,000	15.90
	RL(100;0.9;0.1)	30.76	0.74	21.67	3.92	32	8	6.00	859.88	6.75	2,000	16.46
Quicksort(9)	Exact	54.00	—	31.72	5.90	54	—	160.00	261,156.00	120.00	1,081,621	494.00
	MC	37.68	2.82	25.14	2.34	46	1	1.00	1,772.00	14.00	2,000	15.54
	MCTS(5)	51.74	2.14	42.54	6.91	53	2	32.00	535.50	4.50	2,000	18.24
	RL(100;0.9;0.1)	37.64	2.74	26.05	2.50	44	1	1.00	796.00	6.00	2,000	15.96
TSP(5)	Exact	66.00	—	49.74	11.58	66	—	808.00	3,583.00	12.00	15,018	53.00
	MC	66.00	0.00	37.15	10.57	66	50	3.34	631.02	6.66	2,000	22.90
	MCTS(100)	66.00	0.00	41.87	12.83	66	50	28.82	515.88	5.28	2,000	23.96
	RL(100;0.1;0.9)	66.00	0.00	44.77	12.63	66	50	43.44	370.38	4.00	2,000	26.46
Airplan Shifter(10)	Exact	28.00	—	24.97	1.75	28	—	215,040.00	247,217.00	137.00	3,628,800	2,099.00
	MC	28.00	0.00	22.54	1.92	28	50	6.82	274.04	2.74	2,000	22.66
	MCTS(5)	28.00	0.00	24.78	2.00	28	50	140.14	160.96	1.52	2,000	23.74
	RL(100;0.9;0.1)	28.00	0.00	22.99	1.88	28	50	10.64	321.06	3.14	2,000	21.48
Find Entry(6)	Exact	86.00	—	37.13	8.09	86	—	2.00	604,057.00	789.00	660,665	853.00
	MC	57.28	2.88	31.54	5.00	70	1	1.00	661.00	11.66	2,000	27.26
	MCTS(10)	72.22	6.03	47.56	9.16	86	3	2.00	1,584.67	14.12	2,000	30.66
	RL(100;0.1;0.1)	55.52	7.06	37.29	5.45	80	1	2.00	616.00	9.96	2,000	27.18
Gabfeed(7)	Exact	62.00	—	37.53	6.99	62	—	64	714,904.00	776.00	973,049	1,075.00
	MC	56.76	1.72	33.50	6.56	60	4	1.25	1,057.25	16.25	2,000	28.54
	MCTS(20)	60.32	1.88	39.66	8.54	62	12	1.00	1,593.33	23.33	2,000	28.92
	RL(100;0.1;0.9)	58.62	1.79	37.26	7.02	61	5	1.40	1,218.00	17.80	2,000	28.62
LawDB(7)	Exact	48.00	—	33.38	5.47	48	—	240.00	408,546.00	196.00	545,835	262.00
	MC	43.84	1.20	26.62	4.95	46	4	1.00	910.50	8.00	2,000	18.12
	MCTS(10)	46.12	0.92	36.93	5.32	48	3	18.67	1,403.67	13.33	2,000	19.00
	RL(100;0.1;0.9)	45.50	1.09	30.14	5.60	48	3	1.67	1,727.33	15.00	2,000	17.52
Ngram(14)	Exact	44.00	—	30.00	3.16	44	—	2.00	7,174,439.00	3,409.00	7,174,453	3,409.00
	MC	34.54	1.25	20.84	4.98	40	1	1.00	1,970.00	27.00	2,000	26.88
	MCTS(10)	42.52	1.34	31.44	7.42	44	16	2.00	1,109.00	16.38	2,000	30.26
	RL(100;0.5;0.5)	36.32	1.53	26.28	4.74	40	2	2.00	1,327.00	17.50	2,000	27.62
Passwd Checker(4)	Exact	131.00	—	69.47	18.58	131	—	2.00	1,082,397.00	492.00	1,082,401	492.00
	MC	50.86	0.67	24.09	9.50	52	7	1.86	1,957.43	48.86	2,000	49.98
	MCTS(5)	120.62	9.71	75.88	25.65	131	3	2.00	1,451.00	45.67	2,000	62.54
	RL(1;0.5;0.5)	73.78	2.71	42.47	13.49	85	1	2.00	1,622.00	51.00	2,000	62.24
Text Crunchr(7)	Exact	65.00	—	33.16	5.81	65	—	64.00	888,141.00	1,340.00	1,149,877	1,766.00
	MC	54.46	1.94	30.19	4.50	61	1	1.00	1,340.00	24.00	2,000	36.40
	MCTS(10)	63.56	1.11	43.60	7.79	65	4	12.00	1,690.00	33.25	2,000	39.16
	RL(100;0.1;0.9)	57.22	3.55	31.85	5.29	62	7	1.71	1,386.43	23.57	2,000	34.22

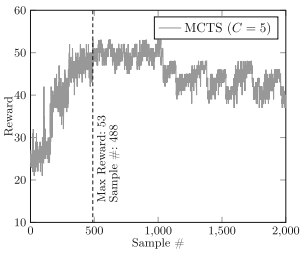


Fig. 3. MCTS quicksort.

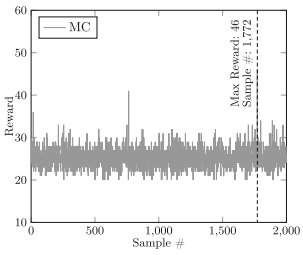


Fig. 4. MC quicksort.

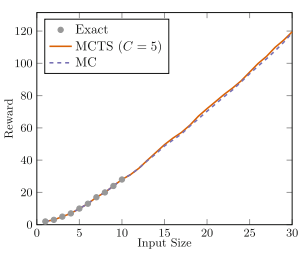


Fig. 5. Airplan shifter.

Scalability: We demonstrate the scalability of MCTS compared to the Exact analysis and how it fares compared to MC. We obtained results for input sizes 1..30 for each example and set a time limit of 1 h for the Exact analysis for each

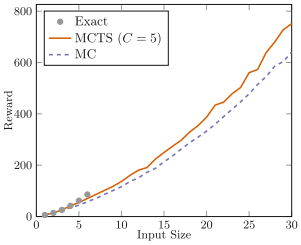


Fig. 6. Find entry.

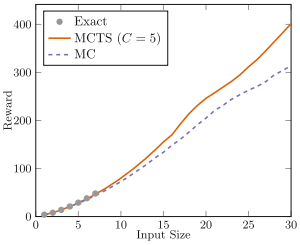


Fig. 7. LawDB.

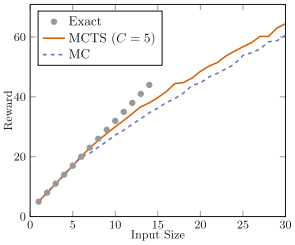


Fig. 8. Ngram.

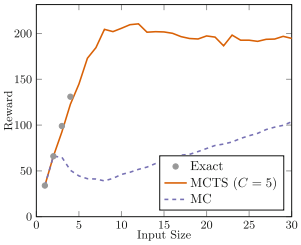


Fig. 9. Passwd checker.

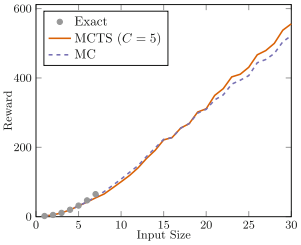


Fig. 10. Gabfeed.

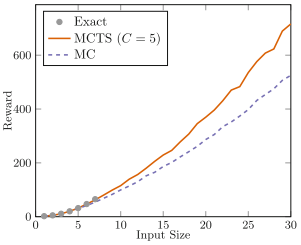


Fig. 11. Text Crunchr.

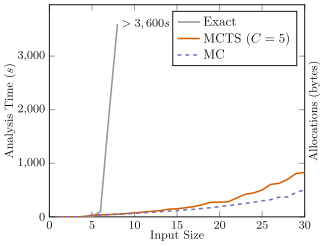


Fig. 12. Analysis time for Text Crunchr.

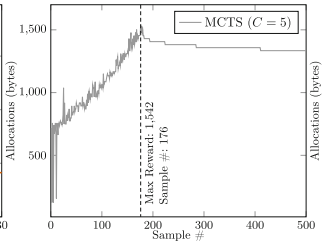


Fig. 13. MCTS WithMi (allocation reward).

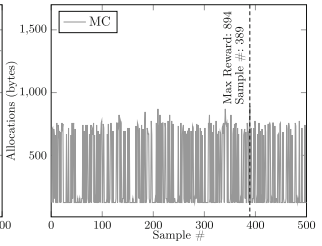


Fig. 14. MC WithMi (allocation reward).

input size; for MCTS ($C = 5$) and MC, we use a budget of 2,000 samples. We ran MC and MCTS 25 times for each input size and report the average maximum reward observed. Figures 5, 6, 7, 8, 9, 10 and 11 show the scalability results.

MC and MCTS scale far beyond the exact analysis and in general—for the input sizes at which Exact analysis does not time out—both techniques are able to find a path with the highest reward with MCTS being slightly better. For greater input sizes, MCTS is better.

Text Crunchr is a representative example of these observations: As shown in Fig. 12, the analysis time of the Exact analysis grows exponentially in the size of the program whereas MCTS and MC grows polynomially; they easily scale to input size 30, but Exact exceeds the 1 h budget at input size 8. Using

MCTS, we found a time complexity vulnerability in Text Crunchr that is due to a component that performs post-processing of word frequency results in $O(n^2)$ time. The vulnerability is that the user controls the input file and can craft input that triggers the worst case behavior. Since MCTS also outputs *concrete test inputs* that expose the path with maximum reward, we used it to mount an actual exploit.

For Password Checker, we noticed that MCTS can find paths with much higher rewards if given a larger budget. Even if the bias parameter is set to aggressively favor the exploitation term, it cannot find deeper paths within the current budget. This explains why the analysis plateaus around input size 8.

Reward Functions for Memory Allocations: We also used the MCTS analysis with a reward function that counts the number of bytes of memory allocations to find costly paths in terms of memory. We analyzed a text compression component of the real-world example, WithMi. It uses a trie for a special encoding of common characters that appear in the text shared by users over a P2P network. Our analysis discovered a vulnerability that would allow a user to maximize the size of the trie and thus the number of allocated nodes, leading to crashing WithMi. Figure 13 shows that MCTS exhibits similar behavior with a reward function that captures allocations; in contrast, Monte Carlo sampling performs poorly (see Fig. 14).

6 Related Work

MCTS was first used in a software engineering context to improve heuristics for a theorem prover [3]. More recent work uses MCTS for optimizing program synthesis [5] and for symbolic regression [13]. MCTS was used before in Java PathFinder [10]. That work considers explicit state model checking and implements a heuristic for deadlock detection.

Meta-heuristics have been used for Worst-Case Execution Time (WCET) analysis—notably genetic algorithms [8]. Unlike all previous approaches, we use symbolic execution and thus instead of searching the concrete input space, we search the symbolic execution tree that encodes the program behaviors. Our search space is thus smaller and we can exploit pruning, which provides a guarantee of convergence to the optimal results. A recent example of using genetic algorithms to find performance bottlenecks is [11]: it uses a fitness function that represents the elapsed time of an application that can be maximized by manipulating input parameters. This is a black-box approach, whereas symbolic execution is by definition white-box.

Another related work uses symbolic execution to find inputs for load testing of software [14]. The technique executes paths with an iterative analysis, starting with a small depth, and expands paths that look most promising in future iterations. We play out executions to their end to find out which paths are more promising and thus have more information to pick the correct paths to favor going forward.

WISE [2] uses symbolic execution and "branch generators" for complexity analysis. Branch generators dictate which branches to take along the worst-case paths. SPF-WCA [6] extends WISE with more general *path policies*, which are history and context dependent. Both WISE and SPF-WCA perform an *exhaustive* analysis for increasing input sizes (to obtain the policies) which may be intractable even at small input sizes. Our analysis uses MCTS to avoid an exhaustive exploration.

7 Conclusion

We presented a symbolic execution method that uses MCTS to guide the search for costly paths in programs, where the cost is defined with respect to space-time consumption. We implemented the approach in Canopy and we evaluated it on complex Java programs. In the future, we plan to increase the scalability of our approach by developing a parallel version and by incorporating domain-specific knowledge, based on decision patterns.

Acknowledgment. We would like to thank the anonymous reviewers for their valuable comments. This material is based on research sponsored by DARPA under agreement number FA8750-15-2-0087. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte Carlo tree search methods. IEEE Trans. Comput. Intell. AI Games **4**(1), 1–43 (2012)
2. Burnim, J., Juvekar, S., Sen, K.: Wise: automated test generation for worst-case complexity. In: IEEE 31st International Conference on Software Engineering, ICSE 2009, pp. 463–473, May 2009
3. Ertel, W., Schumann, J.M.P., Suttner, C.B.: Learning heuristics for a theorem prover using back propagation. In: Retti, J., Leidlmair, K. (eds.) 5. Österreichische Artificial-Intelligence-Tagung, pp. 87–95. Springer, Heidelberg (1989)
4. Kocsis, L., Szepesvári, C., Willemson, J.: Improved Monte-Carlo search. University Tartu, Estonia, Technical report, 1 (2006)
5. Lim, J., Yoo, S.: Field report: applying Monte Carlo Tree Search for program synthesis. In: Sarro, F., Deb, K. (eds.) SSBSE 2016. LNCS, pp. 304–310. Springer, Cham (2016)
6. Luckow, K., Kersten, R., Păsăreanu, C.: Symbolic complexity analysis using context-preserving histories. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 58–68, March 2017
7. Luckow, K., Păsăreanu, C.S., Dwyer, M.B., Filieri, A., Visser, W.: Exact and approximate probabilistic symbolic execution for nondeterministic programs. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE 2014, pp. 575–586, New York, NY, USA. ACM (2014)

8. McMinn, P.: Search-based software test data generation: a survey: research articles. Softw. Test. Verification Reliab. **14**(2), 105–156 (2004)
9. Pasareanu, C.S., Visser, W., Bushnell, D.H., Geldenhuys, J., Mehlitz, P.C., Rungta, N.: Symbolic pathfinder: integrating symbolic execution with model checking for Java bytecode analysis. Autom. Softw. Eng. **20**, 391–425 (2013)
10. Poulding, S., Feldt, R.: Heuristic model checking using a Monte-Carlo Tree Search Algorithm. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, pp. 1359–1366, New York, NY, USA. ACM (2015)
11. Shen, D., Luo, Q., Poshyvanyk, D., Grechanik, M.: Automating performance bottleneck detection using search-based application profiling. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, pp. 270–281, New York, NY, USA. ACM (2015)
12. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of go with deep neural networks and tree search. Nature **529**, 484–503 (2016)
13. White, D.R., Yoo, S., Singer, J.: The programming game: evaluating MCTS as an alternative to GP for symbolic regression. In: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, pp. 1521–1522, New York, NY, USA. ACM (2015)
14. Zhang, P., Elbaum, S.G., Dwyer, M.B.: Automatic generation of load tests. In: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, 6–10 November 2011, pp. 43–52 (2011)

A Cloud-Based Execution Framework for Program Analysis

Daniel Balasubramanian[1(✉)], Dmitriy Kostyuchenko[1], Kasper Luckow[2], Rody Kersten[2], and Gabor Karsai[1]

[1] Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN 37212, USA
{daniel.a.balasubramanian,dmitriy.kostyuchenko, gabor.karsai}@vanderbilt.edu
[2] Carnegie Mellon University, Moffett Field, CA 94035, USA
{kasper.luckow,rody.kersten}@sv.cmu.edu

Abstract. Program analysis is a popular method to determine properties about program behavior, such as execution times and potential security vulnerabilities. One of the biggest challenges faced by almost every form of program analysis is *scalability*. One way to address scalability issues is to distribute the analysis across multiple machines. However, this is not an easy task; designing a distribution framework that is capable of supporting multiple types of program analysis requires careful thought and consideration. This paper presents the cloud-based execution framework that we built for performing distributed analysis of Java bytecode programs. We describe the design decisions that allow this framework to be generic enough to support multiple types of analysis but remain efficient at the same time. We also present a simple, static work partitioning algorithm that we have found to work well in practice and provide benchmarks to show its efficiency.

1 Introduction

Program analysis is a popular method to determine properties about program behavior. It is used by everyday tools, such as compilers, as well as dedicated static and dynamic program analysis engines. One of the main drawbacks of most types of program analysis is that they generally have trouble scaling due to issues such as exponential state-space explosion. Program analysis algorithms may alleviate such problems by introducing either *abstractions* [1] or *heuristics* [2]. Abstractions generally over-approximate the problem and result in coarse-grained answers. Heuristics, on the other hand, try to make the best guess possible, but may result in incorrect answers if the guesses are incorrect.

Even with issues such as exponential state-space explosion, there are often practical benefits that can be gained by *distributing* program analysis [3,4]. If an exploration can be parallelized and distributed across multiple machines, more of the decision space can be explored in a given amount of time. An analysis that requires a week to run on a single machine may be impractical for many

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 139–154, 2018.
https://doi.org/10.1007/978-3-319-92970-5_9

users, but if that analysis can be reduced to a few hours when distributed, then it becomes feasible again.

When adapting an analysis technique for a distributed environment, several issues need consideration. First, one must decide what type of cluster environment will be used to perform the exploration. Supporting only one specific cluster environment is easier than designing a framework that can be reused on multiple types of commodity platforms. More important, though, is the issue of what types of program analysis will be performed. If there is a known, fixed-set of analysis algorithms, limited flexibility in the parallelization framework may be acceptable. If, on the other hand, program analyses are expected to be added over time, the framework must be designed so that these can be accommodated with minimal changes to the overall system.

This paper describes the cloud-based framework that we built for performing program analysis of Java bytecode programs, as well as a static, over-partitioning algorithm for dividing the work. We focus on the design decisions that allow our framework to accommodate multiple types of program analysis. While this paper emphasizes this generality, we have used our framework for scaling symbolic analyses that identify *denial-of-service* (DoS) vulnerabilities [2]. In addition, we provide benchmark results that demonstrate the efficiency of our design.

Our custom framework was written after spending approximately one year performing our program analysis using Apache Spark [5], an open-source cloud framework for distributed data analysis that has also been recently proposed for program analysis and symbolic execution in particular. For this reason, we also present detailed insights and reasons into why the Spark framework was not well-suited for our program analysis and the design decisions to which that experience led us.

The rest of this paper is structured as follows. Section 2 gives background information on symbolic execution and one of the heuristic algorithms that we parallelize. Section 3 describes the design of our parallelization framework. Benchmark results are presented in Sect. 4. We discuss our approach and previous experiences in Sect. 5. We survey related work in Sect. 6, and we conclude in Sect. 7.

2 Background

Our cloud framework was motivated by our desire to parallelize our program analysis techniques [2] that are based on *symbolic execution* for identifying Denial-of-Service (DoS) vulnerabilities in Java bytecode. Symbolic execution is a program analysis technique which executes programs on symbolic rather than concrete inputs, and it computes the program effects as *functions* in terms of the symbolic inputs [6]. The behavior of a program P is determined by the values of its inputs and can be described by means of a *symbolic execution tree* where tree nodes are program states and tree edges are the program transitions as determined by the symbolic execution of program instructions.

For each explored path, symbolic execution maintains a path condition PC – a conjunction of constraints over the symbolic inputs that characterizes exactly

those inputs that follow the path. The feasibility of path conditions is checked using off-the-shelf constraint solvers, as the conditions are encountered during symbolic execution, to detect infeasible paths and to generate test inputs that execute feasible paths. To deal with loops and recursion, a bound can be put on exploration depth.

Our work on SPF-WCA [2] proposed a heuristic for symbolic execution to analyze and identify DoS vulnerabilities in Java bytecode programs. Generally speaking, a DoS vulnerability can be defined as a vulnerability in which a program consumes an unexpectedly large amount of resources while processing an input. For instance, an XML document with recursively defined entities can cause an XML parser to consume an exponential amount of memory while processing this input (also known as an *XML bomb attack*).

SPF-WCA implements a worst-case complexity analysis based on the symbolic execution tool Symbolic PathFinder (SPF). An analyzed algorithm is symbolically executed for increasing sizes of the symbolic input (e.g., the size of the collection to be sorted for a sorting algorithm). This information is precise for small input sizes where exhaustive analysis is still feasible. From those paths, a heuristic is computed that can guide the search for large input sizes. Worst-case paths are collected (with respect to a given cost-model), and a function is fit to these results using regression analysis.

An alternative approach we are also exploring is distributing the analysis of the tool Canopy [7], which is also a DoS vulnerability detection tool based on symbolic execution performed by Symbolic PathFinder. It uses Monte Carlo Tree Search (MCTS) to explore "promising" areas of the symbolic execution tree that seem to maximize the cost. Instead of a systematic exploration of the symbolic execution tree, Canopy uses *sampling* of the paths, i.e. a path is explored from the initial state until termination (or bound is met). MCTS selects how decisions are resolved during sampling. It is well-known that MCTS is easily parallelizable, since the state shared between samples is minimal [8].

Because we began with two program analyses that we wanted to parallelize (pure symbolic execution performed by the SPF tool and heuristic-based symbolic execution performed by the SPF-WCA tool), we designed our framework so that it is not tightly coupled to the particular type of program analysis being performed. This helped minimize the amount of effort needed to add additional techniques (Sect. 3.3 describes how the sampling approach above was implemented).

3 Cloud Framework

The overall architecture of our framework is shown in Fig. 1. At a high-level, the architecture can be separated into two subsystems, one consisting of a *Bridge-Daemon* system, and the other consisting of a *Dispatcher-Worker* system.

The `Bridge-Daemon` system is responsible for initializing the cluster, monitoring its health, distributing work orders, and routing status reports. This system is static for all applications and is agnostic to the analysis being performed. The

`Dispatcher-Worker` system, in contrast, is dynamic and specific to the type of program analysis being performed. It bookends the `Bridge-Daemon` system in that the `Dispatcher` is responsible for defining and partitioning the work, the `Worker` is responsible for executing the work partition, and the `Bridge-Daemon` system is responsible for routing messages between the two. This separation of concerns allows us to keep the fabric of the cluster stable and well-tested even when active development is occurring on the implementation of actual program analysis tasks. To that end, we have worked to keep each of the components relatively simple, easy to reason about, and easy to test. We've also been very careful to make sure that the layers of the system are easily containerized using tools such as Docker for easy deployment to services such as AWS, though so far we have had the luxury of using a networked cluster of dedicated machines.

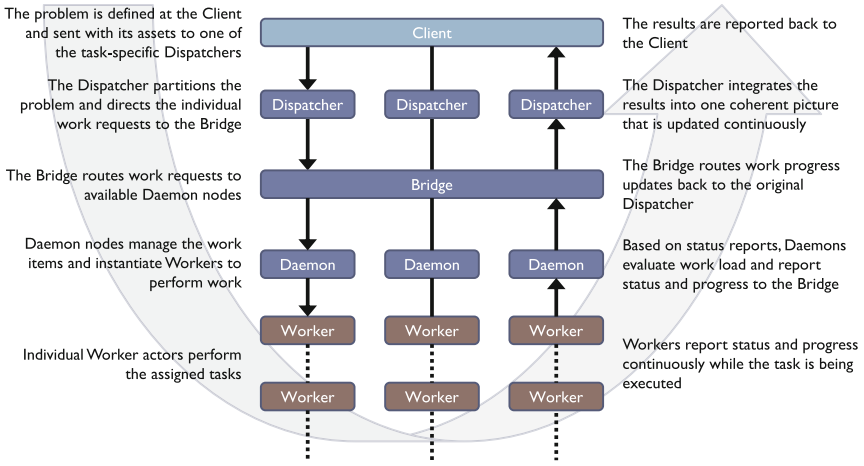


Fig. 1. High-level architecture.

The underlying technology used is Akka Actors [9] defined in Scala, using the internal messaging protocol of Akka and the Kryo object serialization library. Remoting is a problem that is difficult to implement robustly, but is also well-understood. We have chosen to use battle-tested technology in order to remove some uncertainty from our development process and to focus directly on program analysis. Through our work, we have found Akka to be the distributed framework that is both most reliable and least obtrusive.

3.1 Bridge-Daemon Subsystem

The `Bridge-Daemon` system makes up the basic structure of the processing cluster. The `Bridge` component is instantiated on at least two nodes and is responsible for message routing and monitoring. The monitoring is a keep-alive message, handled by the Akka Actors framework, sent periodically from the `Daemon`

to the `Bridge`. The `Bridge` nodes themselves do not contain any application-specific logic; their only function is to coordinate the cluster by maintaining a list of currently available `Daemons` and passing messages between the `Daemons` and `Dispatcher`. This architecture allows `Daemon` nodes to be added and removed in an ad-hoc fashion, without having to define available workers ahead of time.

Each `Daemon` contains a configuration file specifying the address of each `Bridge`. When a `Daemon` node is brought online, it registers itself with each `Bridge` and maintains a keep-alive connection to every `Bridge`. Thus, every `Bridge` is aware of the health of every `Daemon`. A `Daemon` is instantiated on each processing node and is responsible for the coordination of work on that node.

The `Daemons` coordinate the work on a given node by maintaining a queue of outstanding *work items* of a given analysis type. An analysis type is implemented as a custom Java/Scala class defined in the main JAR or loaded from an external JAR at runtime.

The function of the `Daemon` is to receive work requests, examine the task being requested, instantiate a `Worker` for the task, and continuously report the status of work being performed. As with the `Bridge`, the `Daemon` does not contain any application-specific logic. Though it manages the actual work being performed, new types of tasks can be added at any time with minimal or no changes by defining external JARs with additional Java/Scala classes.

3.2 Dispatcher-Worker Subsystem

The application-specific logic is introduced with the `Dispatcher-Worker` system. The `Dispatcher` is responsible for receiving a work request from a client through a REST endpoint. The request is comprised of an archive containing all of the files and parameters needed to execute the task. The `Dispatcher` makes the files needed to do the work available to all `Worker` nodes, determines an appropriate partitioning of the work based on the type of work being performed, and issues a directive to a `Bridge` node to initiate the task using the given parameters.

To perform the partitioning, the `Dispatcher` queries the `Bridge` for the amount of `Daemon` nodes available, which determines the sizes of the partitions. In our symbolic-execution based analyses, for example, the amount of `Daemon` nodes determines the partition depth of the binary decision tree that represents all of the decision instructions in the system under test. The `Dispatcher` then generates all permutations of the subtrees of the decision tree at the determined depth, and shuffles the list in order to help keep the execution time relatively uniform. The shuffled list of subtrees is grouped into task sets, and those task sets are distributed to the available `Daemon` nodes to be queued and executed.

As the task is performed, the `Dispatcher` receives regular status updates from the individual `Worker` nodes in order to compile and report an intermediate picture of the work in progress. The logic of receiving client requests and providing intermediate and final results to the client is implemented in a separate REST service; the `Dispatcher` is implemented such that its only concern is the actual analysis.

The task itself is performed by `Worker` instances. The expectation is that the `Dispatcher` and the `Worker` are created in tandem, so that whatever format the communication between the two takes place in will be understood by both. Thus the directive to start work sent by the `Dispatcher` can be defined in whatever way makes sense for the task; conversely, the status update reports sent by the `Worker` are expected to be understood by the `Dispatcher`. Once a `Worker` receives a request for work, it will begin executing the task while continually making progress reports back up through the system.

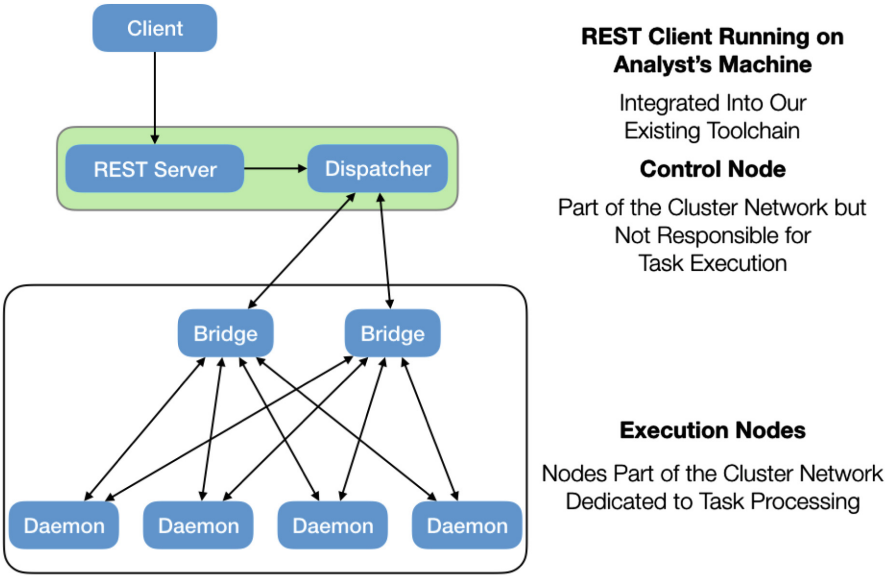


Fig. 2. Network configuration.

The whole process - as illustrated in Fig. 2 - is simple but robust. The `Dispatcher` receives a request from a client over a REST interface. It then determines how the request can be parallelized. The `Dispatcher` then sends a list of specific work items to the `Bridge`, which routes them to the appropriate `Daemon` nodes, with each work item being assigned to a single `Worker` to ensure no duplication of effort. The `Daemon` nodes examine the requests and instantiate specific `Worker` instances. The `Worker` instances, using the parameters in the request, perform the work, while continuously reporting the status of the work and any intermediate results. Those status reports are routed back through the `Daemon` node, back to the `Bridge`, and then back to the original `Dispatcher`. At this point, the `Dispatcher` interprets the results (potentially integrating the separate results into a coherent picture) and makes them available to the client on request. Thus the application-specific logic is kept entirely separate from the cloud processing logic.

In order to support new analysis types, the analyst must define a `Dispatcher` which has knowledge of the analysis being performed and how to partition the problem space. This partitioning depends on the problem: for example in the case of a binary decision tree it might be a series of Yes/No choices that lead to a particular subtree, or in the case of a genetic fuzzing algorithm it might be a set of random seeds. The `Dispatcher` is responsible for sending the problem partitions to the cluster system and receiving results from the executors. The analyst must also define a `Worker` which is able to receive a single problem partition, perform the analysis and report the results back through the cluster system to the `Dispatcher`. These components can be written in any JVM language supporting the Akka framework; currently, that is Scala and Java. The process of implementing new analysis types is largely limited to the implementation of the analysis itself, with practically no attention needing to be paid to the workings of the cluster system.

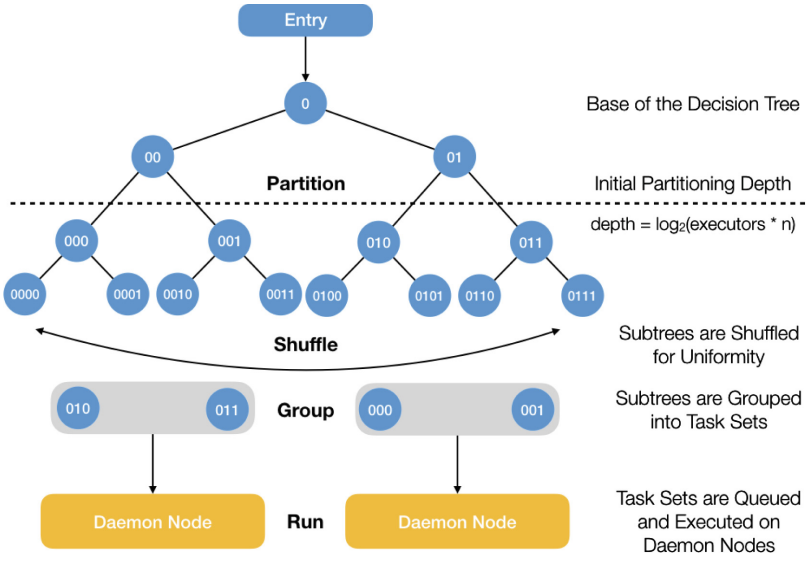


Fig. 3. Static partitioning algorithm for a binary decision tree.

A good example of an application implementation in this system is SPF-WCA. The client requests for an analysis to be performed by sending the system under test and all required files in an archive to the `Dispatcher` REST endpoint. For this particular problem, we partition the decision tree of the application. That is, for every branching instruction encountered, we consider both the true and false options. This creates a binary tree with every node representing a branch in the system under test. At this point, the `Dispatcher` interrogates the `Bridge` regarding how many nodes are available for use, and partitions the entire system decision tree into subtrees to be analyzed separately.

One of the problems we encountered is that the analysis time of a given subtree is wildly unpredictable, and this unpredictability can cause an unequal work distribution. As we are using static partitioning, we have developed a statistical approach to the problem, depicted in Fig. 3 for partitioning a binary decision tree. Because the cost of analyzing each subtree is essentially random, we over-partition the problem space and assign multiple subtrees to the individual workers. As the number of problems approaches infinity, the level of work distribution approaches equality. Once the tasks are defined, the `Dispatcher` assigns a block of tasks to a `Daemon` and sends the task requests to a `Bridge` which forwards the tasks to the defined `Daemon`. The `Daemon` adds the tasks to an internal queue and instantiates `Worker` instances as resources become available. The `Workers` perform the tasks and continually report progress back through the `Daemon`, the `Bridge`, and to the `Dispatcher`. The `Dispatcher` is then able to integrate those results into a single worst-case analysis, which is made available to the client through the same REST endpoint. As the results in Sect. 4 indicate, this static over-partitioning is effective in practice.

3.3 Adding a New Analysis

We present here a practical example of implementing a new type of analysis by looking at Sampling SPF (Canopy), which we have added fairly late in the process. Rather than describe the implementation, we show the real-world code.

We begin (Fig. 4) by creating a `Dispatcher` that partitions the problem space (a binary decision tree) into "Frontiers." These Frontiers are defined as strings composed of '0's and '1's defining the path to a particular subtree. After a set of frontiers is built, the `Dispatcher` divides the set between available workers, creates data structures defining the tasks, and forwards those data structures through the `Bridge`.

For the `Worker` implementation (Fig. 5), we need to create an instance of an existing analysis class that's used for single-machine operation, provide it with configuration and a Frontier from which to begin, and start the analysis. Additionally, we need to respond to status update requests issued regularly by the `Daemon` by querying the analysis class and converting its internal state to a JSON representation.

When the `Dispatcher` receives status updates from the workers, they are added to an overall task status summary which is committed to disk on a regular basis (Fig. 6).

This is a functional minimal integration of a new analysis type, with some boilerplate omitted.

4 Benchmarks and Results

Tables 1 and 2 show benchmark results of exhaustive symbolic execution (SPF) and worst-case analysis (SPF-WCA) applied to a set of classic algorithms used

```
def frontiers(len: Int) = len match {
  case 1          => "0" :: Nil
  case n if n > 1 => (0 to (math.pow(2, (n - 1).toDouble) - 1).toInt).map(_.toBinaryString
     .reverse.padTo(n - 1, "0").mkString).map(s => s"0$s")
  case _          => Nil
}

case dispatcher.Start =>
  val ts = System.currentTimeMillis

  scala.util.Random.shuffle(frontiers((math.log((config.numExecutors * config.
     partitionSize).toDouble) / math.log(2) + 1d).floor.toInt))
    .grouped(config.numExecutors)
    .foreach {
     _.zipWithIndex.foreach { case (frontier, n) =>
       bridge ! DispatcherActivityStart( // activity start message sent through the bridge
         dispatcherId = "samplingDispatcher",
         activityId   = s"${ts}_${File(config.jpfFileName).nameWithoutExtension}
     _$frontier",
         timestamp    = ts,
         host         = Some(workers(n % workers.size)),
         function     = "sampling_spf",
         config       = Seq("frontier" -> frontier, "jpfFileName" -> config.jpfFileName).
     toMap.asJson
       )
     }
  }
```

Fig. 4. Dispatcher Start Implementation

```
case WorkerStart  =>
  startTimeInt = System.currentTimeMillis

  val sw    = new SamplingWorker // SamplingWorker is provided by the analysis library
  workerOpt = Some(sw) // retain durable reference to the SamplingWorker

  taskOptF = Some(Future(sw.runAnalysis(config.frontier, config.toJpfConfig)))
  taskOptF.foreach(_.onComplete(_ => self ! WorkFinished)) // background execution

  parentDaemon ! WorkerStartSuccess( // report work started
    ProgressJson(
      elapsedTime = System.currentTimeMillis - startTimeInt,
      active      = true,
      progress    = JsonObject.empty.asJson
    )
  )

case WorkerStatus =>
  parentDaemon ! taskOptF.map { taskF =>
    // Here we also handle finished and inactive cases, omitted for brevity
    WorkStateActive( // report state of active work
      ProgressJson(
        elapsedTime = System.currentTimeMillis - startTimeInt,
        active      = true,
        progress    = stats2stats(workerOpt.get.getStatus).asJson
      )
    )
  }
```

Fig. 5. Worker Implementation

```
case status: DispatcherActivityStatus => // record results internally
  resultsMap += (status.activityId -> status.status)

case WriteOutput => // persist results to disk
  (workingDir/"output"/"results.json")
    .createIfNotExists(createParents = true)
    .overwrite(resultsMap.toMap.asJson.spaces2)
```

Fig. 6. Dispatcher Results Implementation

for WISE [10]. For comparison, we show the total run times for the same computation being performed on one execution core, 128 cores, and 256 cores, as well as the amount of speedup gained using multiple cores. It is important to note that for our motivating use case (identifying DoS vulnerabilities), the final worst case is typically found long before the entire task completes. Our framework allows us to easily gather intermediate results, and so the advantage of distributed execution is even greater than just the raw speedup. The full run times are provided here as a like-to-like comparison of the performance improvement. The execution times vary so drastically between the two (SPF-WCA is much quicker, but less sound) that it is difficult to find an example that can be practically run on single-core SPF and multi-core SPF-WCA. Regardless, the purpose of these results is to show speedup for two different types of analysis, not to compare different analysis types.

As the results show, the relative amount of speedup varies from problem to problem, but the performance increases relatively linearly, with more complex problems scaling better. This actually works in our favor because the amount of overhead is fixed by the number executors and the number of partitions; as the runtime of a problem on a given worker increases, the effects of system overhead decrease. Further performance increases can be achieved by optimizing the cluster structure, and this remains on our list of future work. And of course,

Table 1. Experimental results for exhaustive symbolic execution on classic algorithms. The speedups compared to the single worker case are shown in parenthesis.

Benchmark	Number of workers		
	1	128	256
`Sorted Linked List` insert	22108 s	188 s (117x)	98 s (225x)
`Red-Black Tree` search	11242 s	167 s (67x)	56 s (200x)
`Quicksort` (JDK 1.5)	14711 s	132 s (111x)	83 s (177x)
`Merge Sort` (JDK 1.5)	19113 s	184 s (103x)	110 s (173x)
`Bellman-Ford`	19507 s	174 s (112x)	110 s (177x)
`Dijkstra's`	16849 s	171 s (98x)	112 s (150x)
`Traveling Salesman`	23409 s	193 s (121x)	99 s (236x)
`Insertion Sort`	10259 s	101 s (101x)	82 s (125x)

Table 2. Experimental results for SPF-WCA on classic algorithms. The speedups compared to the single worker case are shown in parenthesis.

Benchmark	Number of workers		
	1	128	256
`Sorted Linked List` insert	13980 s	129 s (108x)	80 s (174x)
`Red-Black Tree` search	2520 s	36 s (70x)	26 s (96x)
`Quicksort` (JDK 1.5)	5940 s	60 s (99x)	46 s (129x)
`Merge Sort` (JDK 1.5)	2460 s	36 s (68x)	28 s (87x)
`Bellman-Ford`	7260 s	65 s (111x)	38 s (191x)
`Dijkstra's`	13080 s	164 s (79x)	121 s (108x)
`Traveling Salesman`	3420 s	43 s (79x)	33 s (103x)
`Insertion Sort`	2520 s	41 s (61x)	30 s (84x)

the overall execution time can be reduced even more by simply expanding the cluster with additional nodes.

5 Discussion

Our current framework was written after spending approximately one year performing our program analysis with the Apache Spark [5] framework, a popular distributed execution platform built on top of Akka [9]. It is one of the most robust tools for cloud computing, especially scientific computation, though we have found it to be unsuitable for our specific needs. The Spark system is built on the paradigm of making existing computations semantically identical to their distributed versions. Data structures are created in the code, processed with user-defined functions, and the results collected; the concepts are identical to local computation, and the work distribution and result generation across the cluster occurs transparently.

Our implementation of exhaustive SPF exploration using Spark and Apache YARN [11] followed the idiomatic Spark model: we partitioned the decision space of the program as described earlier, encoded those partitions as BitSets denoting the choices needed to be made in order to reach the head of the desired subtree, and used that dataset as input to our process. The function that we mapped over the dataset was the actual exhaustive SPF exploration, with the output being a set of statistics for the explored subtree. We then collected those results, integrated them into a single picture, and reported them back to the user. A conceptually sound implementation, though in practice we encountered difficulties:

1. The SPF exploration process of the symbolic execution tree is a long-running function with unpredictable runtimes for different subtrees. This unpredictability introduced a large amount of uncertainty into the overall runtime

of the analysis, pinning it to the runtime of the longest-running subtree. It is possible to return intermediate results in a SPARK exploration, but the only practical way to do so is to return the results of the individual tasks, limiting the granularity of intermediate data to the overall number of defined tasks. Our tool resolves this problem by adding reporting of intermediate results at arbitrary points during task execution. This allows us to obtain an entire picture of the current state of execution even if the execution, or even an individual task, has not yet completed.

2. Another factor influencing our decision was the high level of abstraction provided by the SPARK framework, which is a double-edged sword. While it was easy to work with, we had limited visibility into the underlying components and the debugging process, while feasible, was slow and tedious. Any imperfections in a tool could have several consequences depending on the mode of failure:
   (a) The particular exploration could have returned bad data; this was essentially impossible to debug, as we had very limited insight into the exploration as it was occurring due to SPARK's high level of abstraction. Our tool allows us to control execution at a low level, with the data from the analysis being available for examination immediately, or being returned to the `Dispatcher` with a clear chain of custody.
   (b) The particular analysis could have crashed with an exception, which would be detected by SPARK and logged, but the actual process of extracting and interpreting logs was more time-consuming than we preferred. In contrast, our tool reports analysis failures directly to the `Dispatcher` and, most crucially, ties them to the input data that caused the failure. This allows for easier and more focused troubleshooting, and even automated recovery.
   (c) The particular analysis could stall, which was a case not readily distinguishable from a normal long-running analysis. This resulted in an analysis that would remain in a partially-completed state for an arbitrarily long duration. Due to the continuous reporting of analysis state, our tool lets us determine if the analysis is still making progress or if it has entered deadlock. This progress reporting is not limited; since the `Workers` have direct access to the running analysis classes, and our reporting format can contain arbitrary data, we are free to include any useful metadata regarding the status of the analysis.
3. We found that cluster performance degraded for large, long-running tasks running on multiple executors. Some of the reasons were due to a great multitude of configuration settings whose effects on our distributed computation were not obvious without trial and error. The customization and configuration of the cluster turned into a persistent and considerable cost. While our tool shifts a lot of the configuration burden from an intermediate layer like YARN directly to us, we have found that we are able to more clearly and easily reason about the impact of the configuration changes we make.

Overall, we found ourselves spending more time debugging the system than developing it, and decided that because the tasks we were executing were long

running, resource-intensive, and having complex failure modes, that we would be better served by a system which provided a more detailed picture of the internal state of the work being performed. Thus we descended one level on the abstraction hierarchy and built something that looks like a stripped-down version of Spark, with node management, status reporting, health monitoring, and work distribution being handled by us. This provided us with complete and easily accessible information regarding the cluster state (because we manage it), the status and distribution of the work (because we control it), and any intermediate result reporting (because we can send anything whenever we want).

6 Related Work

Many solutions exist in the literature for parallelization of specific program analysis tools/techniques. These papers were highly influential to our work and their specific solutions are discussed here for comparison. While in some cases these dedicated distribution implementations may perform slightly better than our framework, the solution we present is generic in the sense that it allows for simple extension with new program analysis techniques. It achieves this generality while maintaining most of the performance, as evidenced by the near-linear speed-up reported in Sect. 4.

A solution for the parallelization of Symbolic PathFinder (SPF) also exists [12].The technique is named Simple Static Partitioning. It first performs an initial shallow symbolic execution, during which path conditions are collected. It then uses a heuristic algorithm to create a static partitioning that it believes, based on the collected path conditions, will be a reasonable distribution of work. The algorithm favors partitioning with respect to commonly used variables and simple (efficiently solvable) constraints. It provides a performance improvement of up to 90% of the ideal parallel speedup.

Cloud9 [13] is a framework for parallel symbolic execution. It runs on commodity hardware and is based on KLEE [14]. In Cloud9, each worker has a queue of jobs, corresponding to unexplored branches of the symbolic execution tree. A load-balancer instructs workers with a high load (long job queue) to send jobs to workers with a low load (short job queue). As balancing is done dynamically, this approach does not require static-partitioning of the search space. Since finding a balanced partitioning is not statically decidable, this approach prevents the overall system having to wait for the worker processing the largest subtree to finish.

A similar approach to distributed concolic execution is taken for LIME Concolic Tester (LCT) [15]. In LCT, it is not inputs that are distributed to workers but rather constraints. While these are larger in size, the argument is that in that case, expensive constraint solving will be done on the side of the available worker.

In SAGE [16], new inputs generated by concolic execution can also be distributed over local threads or to other machines.

In Mayhem [17], multiple symbolic execution engines run in parallel on the same machine. A new engine is instantiated each time execution forks.

Platform as a Service (PaaS) infrastructure has also been proposed for performing distributed verification [4]. In that work, CPAchecker, an open-source Java-based framework for software verification, was ported to the Google App-Engine. The PaaS infrastructure provides benefits like automated scaling and load balancing, but also imposes restrictions, such as access to a specific set of Java classes and the inability to load native libraries. Notably, changes had to be made to the actual verification framework to run it on the Google App-Engine. In contrast, our framework does not impose these types of restrictions, and can be configured to run on various types of Infrastructure as a Service (IaaS), such as Microsoft Azure or Amazon EC2.

Another closely related work, named CloudSDV, does distributed driver verification using Microsoft Azure [3]. One difference is that we use a static, over-partitioning algorithm to statically assign work items to available nodes, while CloudSDV uses a work queue from which individual worker nodes discover and consume verification tasks. CloudSDV also uses the Microsoft Azure APIs, whereas we rely on the Akka Remoting and Akka Cluster Java-based libraries for communication and cluster management tasks.

7 Conclusions

This paper presented a cloud-based framework for performing distributed analysis of Java bytecode programs. We described the architecture and design decisions that allow new analyses to be added relatively easily while at the same time offering good performance. Our benchmarks show the efficiency of the framework and architecture with different types of program analysis. We also described a static work-partitioning algorithm that provides good results in practice and offers a simple but effective alternative to more complex dynamic partitioning algorithms.

We are continually adding new analysis tools to the framework (such as fuzzing) in order to enhance our ability to locate potential vulnerabilities in Java bytecode. Another addition that we are researching is the automatic evaluation of the complexity of the system under test in order to enable automatic complexity scaling in correspondence with the resources available to the cluster. Finally, we are continuing to develop our deployment and monitoring tools to allow our framework to function as a complete turn-key system that can be easily deployed on generic distributed execution platforms such as Amazon AWS, and we are planning on making our tools open-source in the near future.

Acknowledgment. This material is based on research sponsored by DARPA under agreement number FA8750-15-2-0087. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

1. Jeannet, B., Miné, A.: Apron: a library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_52
2. Luckow, K., Kersten, R., Psreanu, C.: Symbolic complexity analysis using context-preserving histories. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 58–68, March 2017
3. Kumar, R., Ball, T., Lichtenberg, J., Deisinger, N., Upreti, A., Bansal, C.: CloudSDV enabling static driver verifier using microsoft azure. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 523–536. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_33
4. Beyer, D., Dresler, G., Wendler, P.: Software verification in the Google app-engine cloud. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 327–333. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_21
5. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)
6. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
7. Luckow, K., Pasareanu, C., Visser, W.: Monte Carlo Tree Search for Finding Costly Paths in Programs. In submission
8. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte Carlo tree search methods. IEEE Trans. Comput. Intell. AI Games **4**(1), 1–43 (2012)
9. Haller, P.: On the integration of the actor model in mainstream technologies: the Scala perspective. In: Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions. AGERE! 2012, pp. 1–6. ACM, New York (2012)
10. Burnim, J., Juvekar, S., Sen, K.: WISE: automated test generation for worst-case complexity. In: Proceedings of the 31st International Conference on Software Engineering, ICSE 2009, Vancouver, Canada, 16–24 May 2009, pp. 463–473 (2009)
11. Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., Baldeschwieler, E.: Apache hadoop yarn: yet another resource negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC 2013, pp. 5:1–5:16. ACM, New York (2013)
12. Staats, M., Pasareanu, C.S.: Parallel symbolic execution for structural test generation. In: Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, 12–16 July 2010, pp. 183–194 (2010)
13. Bucur, S., Ureche, V., Zamfir, C., Candea, G.: Parallel symbolic execution for automated real-world software testing. In: Proceedings of the Sixth Conference on Computer Systems, EuroSys 2011, pp. 183–198. ACM, New York (2011)
14. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX conference on Operating systems design and implementation, OSDI 2008, pp. 209–224. USENIX Association, Berkeley (2008)

15. Kähkönen, K., Saarikivi, O., Heljanko, K.: LCT: a parallel distributed testing tool for multithreaded Java programs. Electron. Notes Theoret. Comput. Sci. **296**, 253–259 (2013)
16. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: whitebox fuzzing for security testing. Queue **10**(1), 20–27 (2012)
17. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy, SP 2012, pp. 380–394. IEEE Computer Society, Washington, DC (2012)

Cross-Architecture Lifter Synthesis

Rijnard van Tonder(✉) and Claire Le Goues(✉)

Carnegie Mellon University, Pittsburgh, USA
{rvantonder,clegoues}@cs.cmu.edu

Abstract. Code translation is a staple component of program analysis. A lifter is a code translation unit that translates low-level code to a higher-level intermediate representation (IR). Lifters thus enable a host of static and dynamic analyses for such low-level code. However, writing a lifter is a tedious manual process which must be repeated for every architecture an analysis aims to support. We introduce cross-architecture lifter synthesis, a novel approach that automatically synthesizes lifters for previously unsupported architectures. Our insight is that lifter synthesis can be bootstrapped with existing IR sketches that exploit the shared semantic properties of heterogeneous architecture instruction sets. We show that our approach automates a significant amount of translation effort for a previously unsupported instruction set, and that it enables discovery of new bugs on new architecture targets through reuse of an existing IR-based analysis.

1 Introduction

Intermediate representations (IRs) are a staple component of compilers [20,23] and program analyses [5,8,11,18]. Code translation can generate programs in an IR from high level source languages (e.g., compilers) or low level machine code (e.g., decompilers). A *lifter* is a code translation unit that emits a higher level, architecture-agnostic intermediate representation of architecture-specific lower-level code. Lifters are central to low-level code analysis because they enable reuse of architecture-agnostic analyses at the IR level (e.g., taint analysis, constraint generation) [14,22], and provide essential high level abstractions for program analysis (CFG and function recovery) [9].

However, writing the translation layer for an IR is onerous, requiring manual translation of architecture-specific instructions (e.g., for x86, ARM, MIPS) to the target IR while preserving the native semantics. Modeling the semantics of a new instruction set requires an engineer to consult instruction manuals numbering up to 1,000s of pages per architecture [14,15]. Recent work raises the importance of automating the lifting process [14]. In our own past work, we identify the potential to reuse existing analyses in the IR for new architectures,[1] but are faced with the undesirable prospect of writing new lifters from scratch.

[1] e.g., https://opam.ocaml.org/packages/bap-warn-unused/bap-warn-unused.1.3.0.

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 155–170, 2018.
https://doi.org/10.1007/978-3-319-92970-5_10

We propose a novel synthesis technique to automate the lifting translation process, with a goal of producing an IR program usable for further program analysis (e.g., to find bugs). At a high level, our technique uses inductive synthesis over finite input-output pairs of native instructions to infer semantically equivalent instructions in the IR. We verify the correctness of synthesized instructions by executing the IR (under associated operational semantics) and comparing computational events with that of native execution. Our approach learns *sketches* (templates) from existing IR instructions, that then drive synthesis. Two key insights enable our synthesis approach. First, software exhibits a "natural" property: code structure is repetitive and predictable [16]. Instruction architectures are inherently heterogeneous, but they share similar semantic operations (e.g., move instructions, arithmetic operations). Our approach mines sketches from existing IR programs which preserve this underlying shared semantics. Moreover, because instructions are not distributed uniformly (e.g., move instructions are more common) [6,16], our approach (1) extends across heterogeneous architectures and (2) achieves high translation coverage. Second, we parameterize synthesis by exploiting statement structure to produce an efficient search.

Prior work only partially addresses the challenge of automatic lifting. Hasabnis et al. [13,14] observe the forward translation of compiler IR (GCC's RTL) to assembly code and produce an inverse mapping from assembly back to the original RTL IR. However, this approach requires a forward translation from the IR to assembly for *each* architecture. This approach is impossible if no such translation exists (typical for low-level IRs, which lift directly from assembly [8,22]). Related synthesis approaches automate discovery of symbolic instruction encodings from input-output pairs [10,15]. By contrast, we address the unique challenge of cross-translating the semantics of instructions to another target language (IR) that supports additional program analysis abstractions (e.g., taint analysis, control flow recovery, function recovery). Recent work in program synthesis has proposed the notion of exploiting existing code for scaling synthesis [6]. To the best of our knowledge, we are the first to demonstrate these ideas toward practical, real-world application by enabling automatic lifter synthesis. Our contributions are as follows:

- **Automatic Lifter Synthesis.** We introduce a technique for automatically synthesizing language translation components that lifts low-level code to an IR. We demonstrate that lifter synthesis enables cross-language translation, allowing analysis reuse on previously unsupported architectures.
- **Learning Synthesis Templates.** We show that mining sketches is effective for translating across heterogeneous instruction architectures. Mining sketches (a) preserves shared semantic properties across architectures and (b) scales synthesis by efficiently constraining the candidate sketch search space.
- **Experimental Evaluation.** We validate our approach by synthesizing a lifter for MIPS, a previously unsupported architecture in the Binary Analysis Platform.[2] On average, the synthesized lifter successfully translates

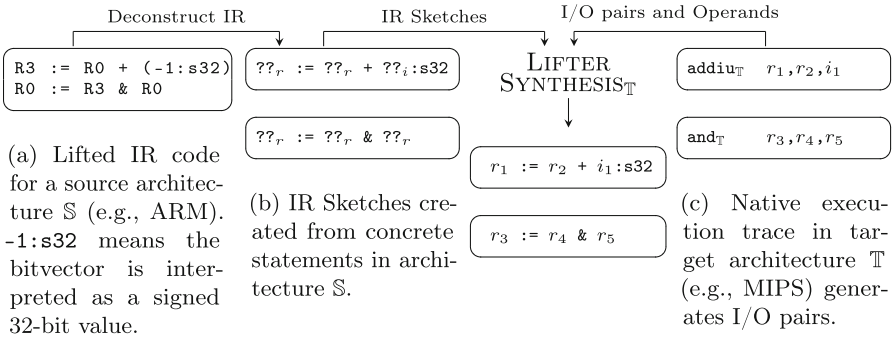
[2] Available at https://github.com/BinaryAnalysisPlatform/bap.

84.4% of instructions to IR, across 28 binaries. Our technique complements additional strategies for lifting the remainder of unlifted instructions (e.g., manually, or with more aggressive synthesis exploration). The synthesized lifter allows a previous IR-based analysis to discover 29 new bugs in binaries for the previously-unsupported architecture.

– **Implementation.** We release our tool and results at https://github.com/squaresLab/SynthLift.

2 Overview and Problem Definition

We formulate IR translation as a syntax-guided synthesis problem [4]. We bootstrap the approach by obtaining an initial set of programs in the IR translated by some existing lifter targeting some other architecture/instruction set (e.g., x86 or ARM).[3] We mine these IR programs to turn concrete program fragments into *sketches* for use in synthesis. Given an unsupported architecture (for which we do not yet have IR translation rules), we (A) collect input-output pairs observed during native execution, and then (B) apply inductive inference over those sketches to discover IR program fragments that satisfy those pairs. We use the oracle-guided inductive synthesis [4] principle to invalidate candidate program fragments using ground-truth input-output pairs.



(a) Lifted IR code for a source architecture 𝕊 (e.g., ARM). `-1:s32` means the bitvector is interpreted as a signed 32-bit value.

(b) IR Sketches created from concrete statements in architecture 𝕊.

(c) Native execution trace in target architecture 𝕋 (e.g., MIPS) generates I/O pairs.

Fig. 1. Synthesizing IR from sketches and I/O pairs.

Overview. Our goal is to use existing IR terms translated from instructions in a source architecture 𝕊 (like ARM) to synthesize satisfying IR translation rules for instructions of a new target architecture 𝕋 (like MIPS). The first step of lifter synthesis deconstructs concrete IR terms (Fig. 1b) from previously lifted code in source architecture 𝕊 (Fig. 1a). Program sketches are syntactic templates that define the search space for synthesis. A sketch is a partial implementation

[3] Note that this is not a limiting assumption on generalizing the technique: an existing, functional IR implies at least one existing translation layer implementation, as is the case with, e.g., REIL [8] LLVM [19], VEX IR [21].

of a program with missing expressions called *holes* [7]; we denote holes by `??` in Fig. 1b. There are two types of holes in our IR sketches: variables, denoted by $??_r$, and immediate bit vector values, denoted by $??_i$ (respectively corresponding to registers and immediate values in the machine architecture).

The second step of synthesis (Fig. 1c) collects concrete input-output pairs, instruction operands, and instruction opcodes from the target architecture $\mathbb{T}$ that we want to lift. In the example, the target architecture is MIPS.[4] We generate traces of input-output pairs by dynamically executing one or more native MIPS instructions. We use the LLVM disassembler to obtain *static* instruction information:: their opcodes, syntactic register names, and immediate values.[5] Static values denoting operands are converted to symbolic IR variables, which we denote in the example by r_x and i_x respectively (x is fresh).

The LifterSynthesis$_{\mathbb{T}}$ procedure then enumerates candidate IR sketches and fills operand holes with the target $\mathbb{T}$'s register and immediate values operand, respectively. The procedure seeks an IR instruction and operand assignment that satisfies all dynamic I/O observations for the native instruction in $\mathbb{T}$ when executed according to the IR's operational semantics. When successful, synthesis produces a *lifter rule* that translates native instructions to the IR for the target $\mathbb{T}$.

Translation Substitution. The synthesis procedure in Fig. 1 identifies IR statements whose *semantics* (specified in Sect. 3) match the input-output pairs of native execution translation rules. For example, an `addiu`$_{\mathbb{T}}$ operand with opcodes $\langle r_1, r_2, i_1 \rangle$ map to an IR statement $r_1 := r_2 + i_1$ `:s32` (note that syntactic register and immediate values are both converted into proper typed values when translated into the IR). Translation binds concrete values to IR operand variables r_x, i_x positionally (Fig. 2).

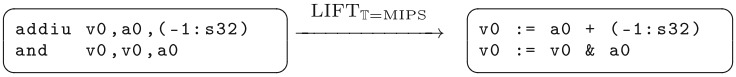


Fig. 2. Lifting to a target $\mathbb{T}$ (MIPS)

In general, we do not know the correct order for applying operands obtained from disassembly; we consider permutation of operands during synthesis in Sect. 4.

Restricting the Synthesis Search Space. The syntactic structure of instructions from native execution allows us to prune the search space of sketches. Fig. 1c gives an intuition: the IR sketch $??_r$ `:=` $??_r$ `&` $??_r$ won't be considered for the `addiu`$_{\mathbb{T}}$ $\langle r_1, r_2, i_1 \rangle$ MIPS instruction because the IR does not use an immediate value. In practice, we find that pruning reduces the set of valid candidate sketches to 83% per native instruction, on average.

[4] https://www.mips.com/products/architectures/mips32/.

[5] We use LLVM for convenience–dynamic binary instrumentation techniques can similarly provide instruction operands and opcodes.

Problem Scope. Our approach synthesizes instructions including arithmetic operations, bitwise operations, and conditional jumps. We do not consider the details of CPU-specific memory models and modes (e.g., concurrency, memory segments, or privileged instructions). While important, these aspects do not directly support the goal of modeling the essential dataflow properties of instruction semantics in the IR. Extension of the IR to additional architecture-specific memory or permission models is possible, but we leave this consideration for future work. For simplicity, we have demonstrated a one-to-one translation of native instruction to IR instruction, whereas IRs are typically designed to represent a single native instruction in one or more IR instructions. We discuss one-to-many translation in Sect. 4.

3 Synthesis Model

We perform oracle-guided synthesis of IR translation using dynamic execution traces of native instructions for a target architecture $\mathbb{T}$. For simplicity of introducing the model, we consider only one iteration of verifying instruction correctness. In one iteration, our goal is to check whether a sequence of *events* produced during a single step of execution of a native instruction is syntactically equal to the sequence of events produced by a executing a *translation* of the native instruction to the IR. Our model assumes a sequential running process, i.e., executing a native instruction is uninterruptible, and memory cannot be modified by concurrent processes. Further, we assume instruction output is invariant under the same inputs. Our assumptions are consistent with the goal of tracking dataflow properties of instruction semantics (e.g., taint analysis, constraint generation), as well as those underlying previous work [10,15]. In this section we introduce the program model and operational semantics for comparing IR and native execution. We use the BAP IR [1,3], which performs competitively relative to other IRs [17]. However, the approach generalizes under the synthesis model and assumptions presented in this section.

3.1 Comparing Executions

Program Model. The execution context of a program is modeled by state σ. Both native and IR instructions operate on a state σ that comprises a memory μ and variable bindings Δ. Memory μ is modeled by a partial function from addresses to values $nat \rightarrow int$. Variable bindings Δ is modeled by a partial function from variable names to values $var \rightarrow int$.

Events. A sequence of events reify the effect of executing an instruction. Events generated during native execution serve as the ground truth oracle for synthesis. We denote events on registers (including flags) by a 4-tuple $\langle action, \texttt{REG}, reg, value \rangle$. An *action* may be either a read operation `R` or a write operation `W`. We use a syntactic value `REG` to disambiguate events on registers from those on memory. A register *reg* may be any syntactic term corresponding to a register for a given architecture (e.g., `EAX` for the x86 architecture). The

value is a bitvector with a word size for a given architecture. We denote events on memory by a 4-tuple $\langle action, \texttt{MEM}, addr, value \rangle$. Actions on memory are the same as for registers. A read action on memory reads a bitvector *value* from a nonnegative address *addr*. A write action on memory writes a bitvector *value* to a nonnegative address *addr*. All events are syntactic elements; we say $e_1 = e_2$ if an event e_1 is syntactically equal to e_2.

Comparing Events. For every instruction executed in the trace of the native Architecture $\mathbb{T}$, a single native instruction $\mathcal{I}_\mathbb{T}$ in state $\sigma_\mathbb{T}$ produces a sequence of events $\mathcal{E}_\mathbb{T}$. We denote the execution step by $\langle \sigma_\mathbb{T}, \mathcal{I}_\mathbb{T}, \varnothing \rangle \overset{\mathbb{T}}{\rightsquigarrow} \langle \sigma'_\mathbb{T}, -, \mathcal{E}_\mathbb{T} \rangle$. For convenience, we define a function $\texttt{step}_\mathbb{T}$ that returns the sequence of events after executing the instructions: $\mathcal{E}_\mathbb{T} = \texttt{step}_\mathbb{T}(\sigma_\mathbb{T}, \mathcal{I}_\mathbb{T})$.

Next, consider execution for IR in an architecture-agnostic language *IR*. Our goal is to generate a sequence of events $\mathcal{E}_{IR}$ which is equivalent to $\mathcal{E}_\mathbb{T}$ by executing a logical instruction (comprising one or more IR statements) denoted by $\mathcal{I}_{IR}$. We denote an execution step in the IR by $\langle \sigma_{IR}, \mathcal{I}_{IR}, \varnothing \rangle \overset{IR_*}{\rightsquigarrow} \langle \sigma'_{IR}, -, \mathcal{E}_{IR} \rangle$ and define a convenience function $\texttt{step}_{IR}$ that returns the sequence of events after execution $\mathcal{E}_{IR} = \texttt{step}_{IR}(\sigma_{IR}, \mathcal{I}_{IR})$.

Executing an IR instruction requires an initial state σ_{IR} that simulates the native architecture state $\sigma_\mathbb{T}$. We introduce a function α_{IR} that resolves register and memory values from the trace, and maps these values to the initial IR execution state, i.e., $\sigma_{IR} = \alpha_{IR}(\sigma_\mathbb{T})$.

We now define an equivalence relation of execution $\sim$ as equal event sequences generated from in-tandem single step execution of source and target languages. Let $\texttt{lift}_{IR}$ be the function that translates a native instruction to target IR instructions, where $\mathcal{I}_{IR} = \texttt{lift}_{IR}(\mathcal{I}_\mathbb{T})$. Then synthesis requires:

$$\texttt{step}_\mathbb{T}(\sigma_\mathbb{T}, \mathcal{I}_\mathbb{T}) \sim \texttt{step}_{IR}(\alpha_{IR}(\sigma_\mathbb{T}), \texttt{lift}_{IR}(\mathcal{I}_\mathbb{T}))$$

which simplifies to checking $\mathcal{E}_\mathbb{T} \sim \mathcal{E}_{IR}$. If $\mathcal{E}_\mathbb{T} \sim \mathcal{E}_{IR}$ holds, a synthesis iteration is complete and $\mathcal{I}_\mathbb{T}$ lifts to $\mathcal{I}_{IR}$. We perform multiple such iterations to refine the accuracy of translation, invalidating IR statements that do not satisfy all input-output equivalence constraints. We defer details of the approach and algorithm to Sect. 4.

3.2 Operational Semantics

Native Execution. The semantics of native execution is treated as a black box, allowing us to observe input-output pairs of an instruction execution. We use dynamic instrumentation to record sequences of events during an execution trace. We support tracing with popular instrumentation frameworks QEMU[6] and Pin.[7] Dynamic events on registers, flags, and memory are recorded in the

[6] https://www.qemu.org/.

[7] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

trace and processed to produce $\mathcal{E}_{\mathbb{T}}$, accepted as ground truth. For purposes of synthesis, we synchronize byte ordering (endianness) for $\mathbb{T}$ and IR at the dynamic instrumentation level, if needed.

IR Execution. We use an analysis-based IR to execute synthesized statements according to an operational semantics.[8] The BAP interpreter performs architecture-agnostic execution of IR statements. Figure 3 is a simplified version of the IR grammar. Our work extends the operational semantics and interpreter to generate events during IR execution. For brevity, we elide the rules. The essential changes entail event recording for each rule: variable assignments on registers and reads produce `Write` and `Read` events, respectively. The same follows for memory accesses; the sequence rule appends events for two instructions, and so on. The full IR grammar and operational semantics is available online [1]. As a concrete example, executing the IR statement $R2 := R0$ results in the event sequence $\mathcal{E}_{IR} = [(\texttt{R,REG,R0,0x1}), (\texttt{W,REG,R2,0x1})]$ (where $R0$ initially stores the value 0x1). The production of ground truth $\mathcal{E}_{\mathbb{T}}$ by native execution and $\mathcal{E}_{IR}$ is compared during synthesis iterations to discover IR statements that satisfy the observed input-output pairs. Note that since we perform a synthesis iteration for one native instruction at a time, execution of the IR code is synchronized with native execution. Our operational semantics therefore does not continue execution by advancing a program counter: instead, it iterates through the sequence of IR statements and executes them until the sequence is empty.[9]

4 Synthesis Approach

We now explain how our synthesis approach generates translation rules that lift native instructions to a sequence of IR statements as a function of the following inputs: (A) A unique identifier for the instruction (i.e., opcode); (B) the set of instruction operands (as purely syntactic values, i.e., register names and immediate values); (C) a set of input-output pairs on register and memory; and (D) a set of candidate sketches in the IR.

4.1 Sketches from Term Deconstruction

Our first key insight is that concrete IR terms (generated from existing lifters) preserve semantic properties to correctly synthesize translation rules for new architectures. Our technique deconstructs concrete IR terms to automatically generate the set of sketch candidates for synthesis. The second key insight is that the syntactic values of native instruction operands (register names and immediate values) reduce the set of possible sketch candidates, making synthesis efficient.

[8] Note that IRs may lack a specified operational semantics. Our work emphasizes the importance of using a formally specified IR to enable translation synthesis.

[9] Note that PC-relative instructions, such as jumps, still need access to a program counter variable to enable synthesis. For this, an internal PC is kept in the execution environment.

We deconstruct concrete IR terms to generate sketches. Formally, a sketch is a *partial function* $\lambda h.S$ that accepts a vector of arguments h, or holes, and generates a concrete term S. The arity of S depends on the number of leaf nodes in the AST of the IR term. The input domain consists of two kinds of terms: free variables (e.g., corresponding to registers), and immediate values (i.e., constants). Note that these two kinds of terms correspond to the leaf nodes *var* and *imm* in the IR grammar, respectively (see Fig. 3).

As an example, suppose we encounter the concrete IR statement `R1 := R0+5`. We recursively visit each term in the statement and generate holes for terminal nodes, thereby deconstructing the statement to yield a sketch as follows:

$$[\![\lambda h.S]\!] \stackrel{def}{=} \_{var} := \_{var} + \_{imm}$$

Three holes are created: the first two operands refer to variables, and the third operand refers to an immediate bitvector value. Given a vector of operands $\overline{o} = \langle \texttt{R5}, \texttt{R6}, 2 \rangle$, we can perform a substitution in S:

$$[\![(\lambda h.S)\ \overline{o}]\!] = \texttt{R5} := \texttt{R6} + 2$$

To apply a vector of operators, it must match the arity in S over the number of variables $|vs|$ and immediate values $|is|$. In our example, S has arity 3, partitioned as an *arity pair* $\langle 2, 1 \rangle$ since $|vs|= 2$ and $|is|= 1$. We use Algorithm 1 to generate a set of candidate sketches from a program in the IR by visiting each statement in the program. The function ToSketch in line 4 takes a concrete term $\mathcal{I}_{IR}$ and turns it into a sketch containing holes (all concrete values in leaf nodes are converted to holes). Line 5 obtains the operands of $\mathcal{I}_{IR}$ and partitions them to obtain the arity pair $\langle |vs|, |is| \rangle$. The result of Algorithm 1 produces a partial function Lookup mapping unique arity pairs to a set of candidate sketches.

program ::= *stmt seq*
stmt s ::=
  var := *exp*
| `jmp` *exp*
| `if` (*exp*) (*stmt seq*)
  `else` (*stmt seq*)

exp e ::=
  exp
| *var* — `variable`
| *imm* — `bitvector value`
| *mem*[exp_1] := exp_2 — `memory store`
| *mem*[*exp*] — `memory load`
| exp_1 *binop* exp_2 — `binary operation`
| *unop exp* — `unary operation`
| *cast* : *nat*[*exp*] — `casts`

Algorithm 1: Mine Sketches

Input:
P_{IR} : program in IR.
Output:
Lookup : $\langle |vs|, |is| \rangle \rightarrow \mathcal{S}$. Lookup returns candidate sketches $\mathcal{S}$ for an arity pair (instruction operand sizes)

```
1 MineSketches(P_IR)
2   for BasicBlock_IR ∈ Visit(P_IR) do
3     for Stmt_IR ∈
        Visit(BasicBlock_IR) do
4       λh.S ← ToSketch(I_IR)
5       ⟨|vs|, |is|⟩ ←
          Partition(Ops(I_IR))
6       Lookup ← UpdateMap(
7         Lookup, ⟨|vs|, |is|⟩, λh.S
8       )
9   ret Lookup
```

Fig. 3. Simplified IR grammar

Algorithm 2: Synthesis

Input:
LOOKUP, $\mathbb{T}$: a lookup function produced by MINESKETCHES and trace input $\mathbb{T}$.
$\mathcal{T}$: a dynamic execution trace.
Output:
LIFT$_\mathbb{T}$: *code* $\rightarrow \mathcal{S}_{IR}$: a function that returns a set of valid sketches in the target IR for the given native instruction code.

```
1  SYNTHESIZE(LOOKUP, 𝒯)
2      for ⟨σ_𝕋, ℐ_𝕋, ℰ_𝕋⟩ ∈ 𝒯 do
3          code, ō ← CODE(ℐ_𝕋), OPS(ℐ_𝕋)
4          Ψ ←
             UPDATEMAP(Ψ, code, {ō, σ_𝕋, ℰ_𝕋})
5          ⟨|vs|, |is|⟩ ← PARTITION(ō)
6          𝒮 ← LOOKUP(⟨|vs|, |is|⟩)
7          ℛ ← SYNTHINSN(
8            σ_𝕋, ō, 𝒮, ℰ_𝕋, Ψ(code)
9          )
10         LIFT_𝕋 ←
             UPDATEMAP(LIFT_𝕋, code, ℛ)
11     ret LIFT_𝕋
```

Algorithm 3: Synthesis Iteration

```
1  SYNTHINSN(σ_𝕋, ō, 𝒮, ℰ_𝕋, ψ)
2      ℛ ← ∅
3      for λh.S ∈ 𝒮 do
4          for λh.S_p ∈ PERM(λh.S) do
5              C_IR ← (λh.S_p) ō
6              ℰ_IR ←
                 step_IR(C_IR, α_IR(σ_𝕋))
7              if ℰ_𝕋 ∼ ℰ_IR ∧
8                  VERIFY(ψ, λh.S_p) then
9                  ℛ ← ℛ ∪ {λh.S_p}
10     ret ℛ
11
12 VERIFY(ψ, λh.S)
13     ret ⋀_{⟨ō,σ_𝕋,ℰ_𝕋⟩∈ψ}
14         (ℰ_𝕋 ∼ step_IR(α(σ_IR), (λh.S) ō))
```

Algorithm 4: Lift

```
1  LIFTHELPER(LIFT_𝕋, ℐ_𝕋)
2      λh.S ←
         TAKEFIRST(LIFT_𝕋(CODE(ℐ_𝕋)))
3      ō ← OPS(ℐ_𝕋)
4      I_IR ← (λh.S) ō
5      ret I_IR
```

4.2 Synthesis

We perform syntax-guided inductive synthesis over sketches. The program synthesis problem stipulates that the formula $\forall x.\phi(x, [\![P]\!](x))$ be valid for all inputs x for a synthesized program $[\![P]\!]$ [7]. The formula ϕ relates an input and output specification against a synthesized program $[\![P]\!]$. For an oracle-based, syntax-guided synthesis the general formula is

$$\forall x.\phi(x, [\![P]\!](x)) \equiv \forall x.\ oracle(x) = [\![P]\!](x)$$

for some equivalence relation $=$. In Sect. 3 we defined the equivalence relation for IR and native execution as equivalence of event sequences. In terms of inputs $\langle \sigma_\mathbb{T}, \mathcal{I}_\mathbb{T} \rangle$ from source language $\mathbb{T}$ (as in Sect. 3) and sketches $[\![S]\!]$, we define the *translation synthesis problem* as finding $[\![S]\!]$ subject to:

$$\forall \langle \sigma_\mathbb{T}, \mathcal{I}_\mathbb{T} \rangle.\phi(\langle \sigma_\mathbb{T}, \mathcal{I}_\mathbb{T} \rangle, [\![S]\!](\langle \sigma_\mathbb{T}, \mathcal{I}_\mathbb{T} \rangle)) \equiv \\ \texttt{step}_\mathbb{T}(\sigma_\mathbb{T}, \mathcal{I}_\mathbb{T}) \sim \texttt{step}_{IR}(\alpha_{IR}(\sigma_\mathbb{T}), [\![(\lambda h.S)\ \text{OPS}(\mathcal{I}_\mathbb{T})]\!]) \quad (1)$$

for each unique $\mathcal{I}_\mathbb{T}$. The synthesis algorithm goal is to discover a sketch $[\![S]\!]$ applied to the operands of native instruction OPS$(\mathcal{I}_\mathbb{T})$ such that (1) holds.

Algorithm 2 describes the synthesis process. Input consists of the lookup function produced by MINESKETCHES and trace information $\mathcal{T}$ containing a set of triples $\langle \sigma_\mathbb{T}, \mathcal{I}_\mathbb{T}, \mathcal{E}_\mathbb{T} \rangle$ generated from dynamic executions $\mathcal{E}_\mathbb{T} = \texttt{step}_\mathbb{T}(\sigma_\mathbb{T}, \mathcal{I}_\mathbb{T})$. Functions CODE and OPS in line 2 of Algorithm 2 extracts a unique identifier *code* associated with $\mathcal{I}_\mathbb{T}$, and its operands as a vector $\overline{o}$, respectively. In line 4,

instruction operands $\overline{o}$, initial execution state $\sigma_{\mathbb{T}}$, and computed events $\mathcal{E}_{\mathbb{T}}$ are associated with unique instruction codes in the map Ψ. Candidate sketches $\mathcal{S}$ are obtained from a partition of the instruction's operands (lines 5 and 6).

SynthInsn enumerates through all candidate sketches to find a satisfying assignment of operands that satisfy events. As mentioned in Sect. 2, we cannot assume that the operand order returned by the disassembler guarantees the desired semantics. In Algorithm 3, line 4, Perm generates sketches that permute the order of input operands in $\lambda h.S$. We discuss permutation strategies in Sect. 4.3. Each permuting sketch $\lambda h.S_p$ applies $\overline{o}$ and generates a concrete IR term C_{IR} and executes it to produce $\mathcal{E}_{IR}$. Lines 7 and 8 verify the concrete term satisfies all events for the instruction $\mathcal{I}_{\mathbb{T}}$ observed so far. The check $\mathcal{E}_{\mathbb{T}} \sim \mathcal{E}_{IR}$ short circuits the more expensive Verify check (line 12) as an optimization. Each satisfying sketch is added to the result set $\mathcal{R}$. Valid sketches in the result set are updated in the map $\textsc{Lift}_{\mathbb{T}}$. Algorithm 4 synthesizes $\texttt{lift}_{IR}$ (introduced in Sect. 3) by transforming the $\textsc{Lift}_{IR}$ map into a lookup function.

In summary, using Algorithms 1–4 we fully derive the desired translation $\mathcal{I}_{IR} = \texttt{lift}_{\mathbb{T}}(\mathcal{I}_{\mathbb{T}})$ from initial inputs P_{IR} and $\mathcal{T}_{\mathbb{T}}$:

$$\textsc{Lift}_{\mathbb{T}} = \textsc{Synthesize}(\textsc{MineSketches}(P_{IR}), \mathcal{T}_{\mathbb{T}})$$
$$\texttt{lift}_{\mathbb{T}} = \lambda\, \mathcal{I}_{\mathbb{T}}.(\textsc{LiftHelper}\ \textsc{Lift}_{IR})$$

4.3 Operand Permutations and One-To-Many Translation

The disassembler may return instruction operands in any order to $\textsc{Ops}(\mathcal{I}_{\mathbb{T}})$. We observed that operand order tends to correspond roughly with a left-to-right reading of assembly instruction semantics. For example, an instruction `add R0, 8` corresponding to a semantic expression `R0 = R0 + 8` would disassemble with the operands in the order $\langle \texttt{R0}, \texttt{R0}, 8\rangle$ However, we also observed small discrepancies. For example, memory store instructions in the IR grammar may swap the source and destination operands compared to the disassembled order. Function Perm thus implements a customizable permutation transformation on operands. Though exhaustive enumeration is feasible for small numbers of operands, we have found that only permuting *adjacent* operands proved effective in practice. When we experimented with an exhaustive permutations approach, we observed no increase in successful synthesis. Complexity of trying all adjacency swapping permutations is fast: linear in arity (order $O(|vs|+|is|)$).

Our current implementation synthesizes one-to-many translation by preserving existing one-to-many mappings implemented in current ARM and x86 lifters. This allows synthesis to discover, e.g., conditional branch statements. On the other hand, relying on a rigid mapping may miss sketches such as multiple consecutive assign statements. We leave sketch composition for synthesis to future work.

5 Evaluation

The goal of SynthLift is pragmatic: to synthesize lifter rules for new architectures, alleviating the need to manually translate the majority of instructions. The focus application is to enable existing analyses for unsupported architectures. We target a previously unsupported architecture, MIPS, and show that the synthesized lifter discovers new bugs in commercial off-the-shelf MIPS binaries. Accordingly, we evaluate SynthLift as follows:

- Is SynthLift effective at enabling existing analyses for previously unsupported architectures (Sect. 5.1)?
- What is the speed and accuracy of SynthLift, and what percentage of instructions can SynthLift recover in widely used programs (Sect. 5.2)?
- How well does SynthLift generalize across architectures (Sect. 5.3)?

5.1 Analysis Reuse

We applied an existing taint-based analysis to find new bugs in COTS binaries for MIPS [2]. The analysis checks for cases where results of C library functions are unused. For example, some C POSIX functions are declared with a "warn unused result" attribute that flags warnings at compile time. Our analysis follows taint flows for function return values to detect such bugs in binaries, where source code is not typically available. The analysis looks for cases where the return value is overwritten without being read. We ran the analysis on 30 binaries in the `sbin` directory of a COTS D-Link router. In total, we discovered 29 bugs in 30 binaries; for brevity, we summarize 8 binaries comprising 17 bugs that span a variety of functions handled by the analysis (Fig. 4a). Not shown, we discovered 12 additional bugs across 12 additional binaries for similar functions as in Fig. 4a. 11 binaries did not generate bug reports. We manually inspected analysis results using a decompiler to confirm true positives; where possible, we were able to confirm unchecked values for binaries that have source code (such as `ntpclient`). We encountered two false positives. This happened when return values of two consecutive `malloc` calls are inaccurately tracked in our ABI model (note: the inaccuracy is not due to the synthesized instruction semantics). To consider a large real-world example, we also lifted OpenSSL to recover 86% of instructions, and confirmed that the analysis did not find any bugs.

5.2 Synthesizing the MIPS Lifter

To synthesize the MIPS lifter, we used IR sketches generated from 28 ARM Coreutils[10] binaries, and used 5 programs from the Hacker's Delight benchmarks [24] (compiled to MIPS) to generate dynamic input-output pairs. Coreutils is a set of highly popular command-line utilities and representative of typical programs; Hacker's Delight programs perform a variety of bit-manipulation operations that generate input-output pairs for a diverse set of native instructions.

[10] https://www.gnu.org/software/coreutils/coreutils.html.

Name	#	Functions
† iptables	3	`fwrite`
† ntpclient	1	`send`
† pppd	1	`fwrite`
rdnssd	1	`setsockopt`
speedtest	2	`system, fgets`
timer	2	`read, shutdown`
wakeOnLanProxy	1	`shutdown`
wcnd	8	`system`

(a) # indicates the number of unused return values in COTS MIPS binaries. We show **Function** names for which return values are not used. † indicates that we found evidence of the bug in source code.

%	ARM-IR Sketches	%	x86-IR Sketches
20.9	$\_v$:= $\_v$	11.9	$\_v$:= $\_i$
14.5	$\_v$:= $\_i$	8.6	jmp $\_i$
8.9	jmp $\_i$	5.1	$\_v$:= mem[$\_v$ + $\_i$]
7.0	$\_v$:= mem[$\_v$ + $\_i$]	4.4	mem[$\_v$ + $\_i$] := $\_v$
5.6	$\_v$:= $\_v$ = $\_i$	4.2	$\_v$:= $\_i$ = $\_v$
5.6	$\_v$:= hi : 1[$\_v$]	4.2	$\_v$:= hi : 1[$\_v$]
5.5	mem[$\_v$ + $\_i$] := $\_v$	4.0	mem[$\_v$] := $\_i$
4.6	$\_v$:= $\_v$ − $\_i$	3.9	$\_v$:= $\_v$ − $\_i$

(b) Distribution of single-statement sketches mined from x86 and ARM IR programs. Holes v denote a variable; i, an immediate. We omit bit widths for brevity, though `hi:1` denotes a cast which keeps the high bit of the value (e.g., common for testing IR flags).

Fig. 4. Analysis results and distribution of sketches mined from IR.

End-to-end synthesis (mining sketches, processing traces, and lifter synthesis) takes 58 s. Each native MIPS instruction starts with a set of 29 initial sketches on average (using Algorithm 2 PARTITION and instruction operands). On average, successfully synthesized instructions complete with 2 satisfying sketches (due to commutativity of binary operations). Synthesis converges quickly: Fig. 5, left boxplot, shows that synthesis discovers the final set of satisfying sketches after only two input-output pairs for most instructions. The final set of satisfying sketches verify over thousands of input-output events for typical instructions (Fig. 5). We observe that the distribution of input-output pairs by Hacker's Delight binaries mirror the intuition that common instructions like "load word" (`LW`) represent a disproportionately large part of the programs.

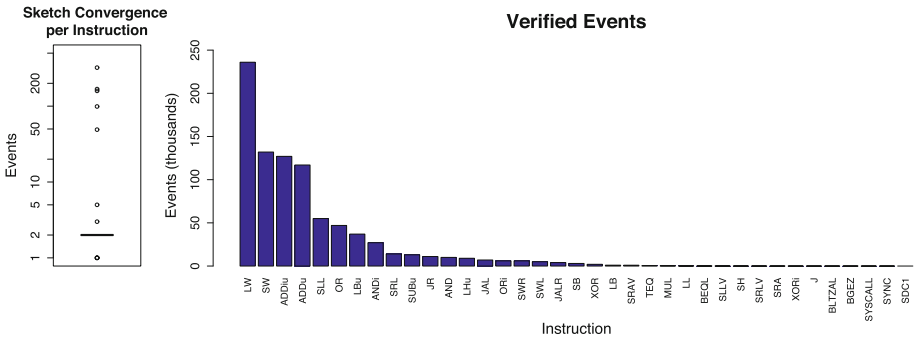


Fig. 5. MIPS Lifter Synthesis. On the left, the number of iterations until synthesis converges on the final set of satisfying sketches over all events. On the right, the number of verified input-output pair events for successfully synthesized instructions.

We ran the synthesized lifter on the 28 MIPS Coreutils binaries in the Debian distribution. To count lifter coverage, we take the percentage of individual native MIPS instructions that fire a translation rule in the lifter. On average, the lifter recovers **84.8%** of instructions. Thus, 15.2% of instructions in the binary could not be lifted (in practice, we substitute NOP instructions in the IR). Synthesis fails when a suitable sketch cannot satisfy the semantics of an instruction. One such instruction is `LUi`, or load upper immediate. To lift this instruction, we ideally want an IR candidate such as $\_var := \_imm << \_imm_2$, where imm_2 is 16. However, such a candidate is never mined from the IR sourced from ARM. In Sect. 5.4 we suggest further improvements to our technique to address such cases.

5.3 Generalizing Across Architectures

BAP currently supports lifting for both the ARM and x86 architectures. To validate our ability to synthesize across architectures, we targeted MIPS by mining sketches lifted from the suite of x86 Coreutils programs. The x86-sourced MIPS lifter synthesized six fewer instructions than the ARM version due to missing a rotation sketch.[11] Interestingly, the x86-sourced lifter recovered the same 84.8% instructions when lifting the MIPS Coreutils test set. Figure 4b suggests why we gain the same utility when synthesizing under different architectures: the eight most frequent sketches for ARM and x86 are very similar, and account for the majority of IR instructions.

5.4 Discussion

Mining versus Manually Specifying. Our approach demonstrates the applicability and feasibility of mining sketches to enable a cross-architecture translation. Figure 4b also suggests that manually specifying a small set of sketches is competitive to mining sketches. However, we observe that manual specification poses additional challenges compared to mining: (a) it is difficult to anticipate exactly the set of sketches to specify; current approaches usually involve a human-in-the-loop who must iteratively estimate or consult specification manuals [10]; (b) the set of effective sketches changed based on how the IR is designed (i.e., different IRs will require different sketch templates); (c) manual specification does not naturally consider similarities of multiple heterogeneous architectures; our summary in Fig. 4b is a first result to show that sketches *do* translate for IRs. Our approach sees manual specification as complementary: mining is an effective approach for revealing initial common sketches (and how the IR-specific design structure relates to sketches), and can automatically discern similarity in e.g., architectures at the IR level. A human-in-the-loop can use this information to make synthesis more effective.

[11] The missing rotation operator is however found in subexpressions of IR statements, but we fail to generate the desired statement $\_var := \_var << \_imm$.

Partial Instruction Set Recovery. We showed in Sect. 5.2 that the synthesized lifter recovers a high percentage (roughly 85%) of instructions in typical binaries. On the other hand, the lifter has a lower rate of coverage for the entire MIPS instruction set, approximately 33 instructions of 45.[12] While a greater percentage of the instruction set is desirable, our goal is to (a) assess whether translation can be synthesized "out-of-box" without specifically considering the target architecture and (b) validate how well existing analyses can operate with a partial lifter synthesized for a new architecture. Our evaluation reveals ample opportunity for improving instruction set coverage (e.g., manually specifying missing sketches) and existing work has shown nondeterminstic approaches, like stochastic search [15] to be effective. At present, our goal is to demonstrate synthesis effectiveness using a tractable method, i.e., using only the set of finite sketches mined from existing rules. We leave the appeal of combining complementary approaches to future work.

6 Related Work

Bornholt et al. [6] propose mining sketches for structure to scale program synthesis. Our work demonstrates the ability to fill this gap by mining IR sketches to scale IR translation over heterogeneous architecture instruction sets. Our work relates generally to syntax-guided synthesis over sketches [4,7]. Related work in inductive synthesis uses I/O pairs to recover x86 semantics as SMT encodings [10,15]. Our approach similarly uses I/O pairs to infer semantics, but targets IR translation for multiple architectures and mines sketches automatically in lieu of manual specification. Hasabnis et al. leverage forward source-to-compiler-IR translation [14] and symbolic execution of compilers [13] to lift low level instructions to the compiler IR. These approaches rely on the existence of a forward translation routine (i.e., compiler) for each architecture, which then reverse the mapping to generate assembly-to-IR rules. In contrast, our approach generalizes to cross-architecture translation using a bootstrapped set of initial candidate sketches and input-output pairs only—no existing translation is required for *each* architecture target. Applications in static binary translation manually translate dynamically executed QEMU instructions to static LLVM IR [9] for multiple architectures; we believe our technique has the ability to automate the translation process. Work on verifying correctness of low level IRs is complementary to the lifter synthesis problem; related techniques can assert correctness of semantics with respect to observed I/O-pairs [10,12] or symbolic equivalence checking [17].

7 Conclusion

We have presented cross-architecture lifter synthesis, a new way to automatically synthesize IR translation rules for new architectures by leveraging existing IR

[12] Using Fig. 5, (excluding instructions `TEQ`, `SYSCALL`, `SYNC`, and `SDC1` which are modeled differently in the trace than actual MIPS semantics), and compared to a simplified MIPS ISA (goo.gl/YUEdiy).

programs. We demonstrated that our approach is effective at recovering a lifter for a new architecture, and provides sufficient instruction coverage to enable analysis reuse and discovery of new bugs. Synthesis could discover more rules by generating candidates over the IR grammar (e.g., using stochastic search [4,15]), or by manually supplying a small number of plausible sketches (rather than manual, per-instruction translation). We further believe our work has further application for discovering semantic relations between different languages: lifter synthesis reveals similar semantic properties across heterogeneous architectures and can distinguish differences when cross-translation synthesis fails. Lifter synthesis opens up new methods for language translation, e.g., by complementing manual processes, and is amenable to automation assistance where sketches can be manually specified (e.g., [10]). Finally, we believe the approach has broad application to IRs generally, including automatic discovery and synthesis of common semantics for IR-to-IR translation.

Acknowledgments. This work is partially supported under NSF grant number CCF-1563797. All statements are those of the authors, and do not necessarily reflect the views of the funding agency. The authors would like to thank the BAP Team for their continued open source development and support for the BAP project.

References

1. BAP IR Operational Semantics (2018). https://github.com/BinaryAnalysisPlatform/bil/releases/download/v0.1/bil.pdf. Accessed 23 Apr 2018
2. BAP Warn Unused Analysis (2018). https://opam.ocaml.org/packages/bap-warn-unused/bap-warn-unused.1.3.0/. Accessed 23 Apr 2018
3. Binary Analysis Platform (2018). https://github.com/BinaryAnalysisPlatform/bap. Accessed 23 Apr 2018
4. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, pp. 1–8 (2013)
5. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: Compiler Construction, pp. 2732–2733 (2004)
6. Bornholt, J., Torlak, E.: Scaling program synthesis by exploiting existing code. Machine Learning for Programming Languages (2015)
7. Bornholt, J., Torlak, E., Grossman, D., Ceze, L.: Optimizing synthesis with metasketches. In: POPL 2016, pp. 775–788 (2016)
8. Dullien, T., Porst, S.: REIL: a platform-independent intermediate representation of disassembled code for static code analysis. In: CanSecWest 2009 (2009)
9. Federico, A.D., Payer, M., Agosta, G.: rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In: CC 2017, pp. 131–141 (2017)
10. Godefroid, P., Taly, A.: Automated synthesis of symbolic instruction encodings from I/O samples. In: PLDI 2012, pp. 441–452 (2012)
11. Gotovchits, I., van Tonder, R., Brumley, D.: Saluki: finding taint-style vulnerabilities with static property checking. In: BAR 2018 (2018)
12. Hasabnis, N., Qiao, R., Sekar, R.: Checking correctness of code generator architecture specifications. In: CGO 2015, pp. 167–178 (2015)

13. Hasabnis, N., Sekar, R.: Extracting instruction semantics via symbolic execution of code generators. In: FSE 2016, pp. 301–313 (2016)
14. Hasabnis, N., Sekar, R.: Lifting assembly to intermediate representation: a novel approach leveraging compilers. In: ASPLOS 2016, pp. 311–324 (2016)
15. Heule, S., Schkufza, E., Sharma, R., Aiken, A.: Stratified synthesis: automatically learning the x86–64 instruction set. In: PLDI 2016, pp. 237–250 (2016)
16. Hindle, A., Barr, E.T., Gabel, M., Su, Z., Devanbu, P.T.: On the naturalness of software. Commun. ACM **59**(5), 122–131 (2016)
17. Kim, S., Faerevaag, M., Junk, M., Jung, S., Oh, D., Lee, J., Cha, S.K.: Testing intermediate representations for binary analysis. In: ASE 2017 (2017)
18. Kinder, J., Veith, H.: Precise static analysis of untrusted driver binaries. In: FMCAD 2010, pp. 43–50 (2010)
19. Lattner, C., Adve, V.S.: LLVM: a compilation framework for lifelong program analysis & transformation. In: CGO 2004, pp. 75–88 (2004)
20. Le, V., Sun, C., Su, Z.: Randomized stress-testing of link-time optimizers. In: ISSTA 2015, pp. 327–337 (2015)
21. Molnar, D., Li, X.C., Wagner, D.A.: Dynamic test generation to find integer bugs in x86 binary linux programs. In: USENIX Security Symposium 2009 (2009)
22. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: IEEE Security and Privacy, pp. 317–331 (2010)
23. Sun, C., Le, V., Zhang, Q., Su, Z.: Toward understanding compiler bugs in GCC and LLVM. In: ISSTA 2016, pp. 294–305 (2016)
24. Warren, H.S.: Hacker's Delight. Pearson Education, London (2013)

Model Checking and Runtime Verification

Counterexample Simplification for Liveness Property Violation

Gianluca Barbon[1(✉)], Vincent Leroy[2], and Gwen Salaün[1]

[1] Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France
gianluca.barbon@inria.fr
[2] Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, 38000 Grenoble, France

Abstract. Model checking techniques verify that a model satisfies a given temporal property. When the model violates the property, the model checker returns a counterexample, which is a sequence of actions leading to a state where the property is not satisfied. Understanding this counterexample for debugging the specification is a complicated task because the developer has to understand by manual analysis all the steps (possibly many) that have provoked the bug. The objective of this work is to improve the comprehension of counterexamples and thus to simplify the detection of the source of the bug. Given a liveness property, our approach first extends the model with prefix/suffix information w.r.t. that property. This enriched model is then analysed to identify specific states called neighbourhoods. A neighbourhood consists of a choice between transitions leading to a correct or incorrect part of the model. We exploit this notion of neighbourhood to highlight relevant parts of the counterexample, which makes easier its comprehension. Our approach is fully automated by a tool that we implemented and that was validated on several real-world case studies.

1 Introduction

Recent computing trends promote the development of hardware and software applications that are intrinsically parallel, distributed, and concurrent. This is the case of service-oriented computing, cloud computing, cyber-physical systems or the Internet of Things. Designing and developing distributed software in this context is a tedious and error-prone task, and the ever increasing software complexity is making matters even worse. Although we are still far from proposing techniques and tools avoiding the existence of bugs in a software under development, we know how to automatically chase and find bugs that would be very difficult, if not impossible, to detect manually.

Model checking [1] is an established technique for automatically verifying that a behavioural model, e.g., a Labelled Transition System (LTS), satisfies a given temporal formula written with temporal logic. When the model violates the property, the model checker returns a counterexample, which is a sequence

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 173–188, 2018.
https://doi.org/10.1007/978-3-319-92970-5_11

of actions leading to a state where the property is not satisfied. Understanding this counterexample for debugging the specification is a complicated task for several reasons: (i) the counterexample can contain many actions, (ii) the debugging task is mostly achieved manually, and (iii) the actions in the counterexample seem to have the same importance even if it is not the case. Note that in this work, we have a specific focus on liveness properties, and more precisely inevitability properties which are one of the classes of liveness properties most used by developers in practice [9].

Our goal in this paper is to simplify the debugging of concurrent systems whose specification compiles into a behavioural model. To do so, we propose a novel approach for improving the comprehension of counterexamples by highlighting some of the actions in the counterexample that are of prime importance, that is, actions that make the specification go from a (potentially) correct behaviour to an incorrect one. These parts of the model correspond to decisions or choices that are of particular interest because they might explain the source of the bug. Once these specific actions have been identified, they can be used for building a simplified version of the counterexample, keeping only actions that are relevant from a debugging perspective.

Our approach takes as input a behavioural model (LTS) describing all possible executions of a system. This LTS can be obtained by compilation from a higher-level textual specification language such as process algebra. Given such an LTS and a liveness property, in a first step, we enhance each state of the LTS model with prefix/suffix information about the actions belonging to the property that have already been or remain to be executed. This enriched LTS is then analysed to identify specific states called neighbourhoods. A neighbourhood consists of a choice between transitions leading to a correct or incorrect part of the model. Those states identify specific parts of the specification that may explain the appearance of the bug and are therefore meaningful from a debugging perspective. Several simplification techniques can be defined on top of this notion of neighbourhood, which aim at removing irrelevant parts of the counterexample and highlighting relevant ones to simplify its comprehension. Our approach is fully automated by a tool that we implemented. This tool was applied on several real-world case studies for evaluation purposes.

The paper is organized as follows. Section 2 introduces behavioural model and model checking. Section 3 presents the technique for computing the LTS enriched with prefix/suffix information. This information is then used for identifying neighbourhoods and building counterexample abstractions from them. Section 4 illustrates our approach on two real-world case studies. Section 5 overviews related work and Sect. 6 concludes this paper.

2 Preliminaries

In this work, we adopt *Labelled Transition System (LTS)* as behavioural model of concurrent programs. An LTS consists of states and labelled transitions connecting these states.

Definition 1. *(LTS) An LTS is a tuple* $M = (S, s^0, \Sigma, T)$ *where* S *is a finite set of state identifiers;* $s^0 \in S$ *is the initial state identifier;* Σ *is a finite set of labels;* $T \subseteq S \times \Sigma \times S$ *is a finite set of transitions.*

A transition is represented as $s \xrightarrow{l} s' \in T$, where $l \in \Sigma$. An LTS is produced from a higher-level specification of the system described with a process algebra for instance. Specifications can be compiled into an LTS using specific compilers. In this work, we use LNT [6] and LOTOS [5] as specification languages and compilers from the CADP toolbox [10] for obtaining LTSs from these specifications. However, our approach is generic in the sense that it applies on LTSs produced from any specification language and any compiler/verification tool. An LTS can be viewed as all possible executions of a system. One specific execution is called a *trace*.

Definition 2. *(Trace) Given an LTS* $M = (S, s^0, \Sigma, T)$*, a trace of size* $n \in \mathbb{N}$ *is a sequence of labels* $l_1, l_2, \ldots, l_n \in \Sigma$ *such that* $s^0 \xrightarrow{l_1} s_1 \in T, s_1 \xrightarrow{l_2} s_2 \in T, \ldots, s_{n-1} \xrightarrow{l_n} s_n \in T$*. A trace is either infinite because of loops or the last state* s_n *has no outgoing transitions. The set of all traces of* M *is written as* $t(M)$*.*

Model checking consists in verifying that an LTS model satisfies a given temporal property φ, which specifies some expected requirement of the system. Temporal properties are usually divided into two main families: *safety* and *liveness* properties [1]. In this work, we focus on a class of liveness properties, called *inevitable execution* properties. Most of the patterns that commonly occur in the specification of liveness properties make use of the inevitable executions. This is the case of the *Response Property Pattern*, that is the most common pattern in [9]. An inevitable execution property states that, given an LTS M and an action l, every trace from the initial state in M presents a transition with the action l. In this work we support *nested inevitable executions*. For instance, a property with two nested actions l_1 and l_2 states that every trace in a given model must exhibit the action l_1 later followed by the action l_2. Note that the two actions do not need to be contiguous in traces. To express nested inevitable executions we define a *nested inevitability operator* using the *Action-based Computation Tree Logic (ACTL)* [8]:

Definition 3. *(Nested Inevitability Operator) Given a sequence of labels* $l_1, \ldots l_n$*, the nested inevitability operator is defined as*

$$\texttt{Inev}(l_1, l_2, \ldots l_n) = A[true_{true} U_{l_1} A[true_{true} U_{l_2} \ldots A[true_{true} U_{l_n} true] \ldots]]$$

where A *and* U *denote the ACTL operators* along All paths *and* Until*, resp.*

A nested inevitable execution property can be semantically characterised by a possibly infinite set of traces t_φ, corresponding to the traces that comply with the property φ in an LTS. If the LTS model does not satisfy the property, the model checker returns a *counterexample*, which is one of the traces characterised by $t(M) \setminus t_\varphi$.

Definition 4. *(Counterexample) Given an LTS* $M = (S, s^0, \Sigma, T)$ *and a property* φ*, a counterexample is any trace which belongs to* $t(M) \setminus t_\varphi$*. A counterexample can be in the form of an elementary trace, which is a trace where states are pairwise distinct, or a lasso, which is a trace* $s^0 \xrightarrow{l_1} s_1 \in T, \ldots, s_{n-2} \xrightarrow{l_{n-1}} s_{n-1} \in T, s_{n-1} \xrightarrow{l_n} s_n \in T$*, such that* $s^0 \xrightarrow{l_1} s_1 \in T, \ldots, s_{n-2} \xrightarrow{l_{n-1}} s_{n-1} \in T$ *is an elementary trace and* $s_n = s_i$ *for some* $0 \leq i < n$*.*

3 Counterexample Simplification

In this section we discuss in detail our approach to simplify counterexamples. Section 3.1 presents the notions of prefixes and suffixes. Section 3.2 describes the algorithm to compute them and enrich the initial LTS. In Sect. 3.3 we identify transitions and we introduce the neighbourhood notion. Section 3.4 presents simplification techniques, focusing on one of them.

3.1 Prefixes and Suffixes

An LTS M is a model representing all possible executions of a system. Given an inevitable execution property φ, our goal is to analyse each state s in M to understand whether all the traces that pass through s satisfy the sequence of actions expressed by φ. To do this we compare the prefixes of traces that reach the state to prefixes of the given sequence. Similarly we compare the suffixes of traces that start from the state to suffixes of the given sequence. Note that in this work we use the symbol "$\cdot$" to denote the concatenation operator for labels and sequences of labels.

Definition 5. *(Sequence of Inevitable Actions) Given an inevitable execution property* $p = \texttt{Inev}(l_1, \ \ldots, \ l_n)$*, the sequence of concatenated labels* $k = l_1 \cdot l_2 \cdot \ldots \cdot l_n$ *of size* $n \in \mathbb{N}$ *is the sequence of inevitable actions that respect the order defined by the nested inevitability operator.*

The sequence of inevitable actions may represent non-contiguous transitions in the model. In order to match traces and prefixes (suffixes, resp.) of traces with the sequence of inevitable actions, we define a matching operator as follows:

Definition 6. *(Matching Operator) Given an LTS* $M = (S, s^0, \Sigma, T)$*, a sequence of labels* $j = a_1 \cdot a_2 \cdot \ldots \cdot a_n$*, a sequence of contiguous transitions* $z = s_1 \xrightarrow{l_1} s_2 \in T, s_2 \xrightarrow{l_2} s_3 \in T, \ldots, s_{m-1} \xrightarrow{l_{m-1}} s_m \in T$*, z is said to match j, written* $j \prec z$*, if there exists integers* $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ *such that* $a_1 = l_{i_1}, \ a_2 = l_{i_2}, \ \ldots, \ a_n = l_{i_n}$*.*

We assign to each state of the LTS the prefixes of the sequence of inevitable actions k obtained up to the state under analysis. To do this, we introduce the notions of *max prefix* and *common prefix*, w.r.t. k. The max prefix is the longest prefix of the k sequence among the prefixes of traces that end in a given state.

The common prefix is the longest prefix of the k sequence that is common to all the prefixes of traces that end in a given state. We define $\mathbb{T}_s^e$ as the set of all the prefixes of traces that end in s and $\mathbb{P}^k$ as the set of all the prefixes of k.

Definition 7. *(Max and Common Prefix) Given an LTS $M = (S, s^0, \Sigma, T)$, a sequence of inevitable actions k, the set $\mathbb{P}^k$ of all the prefixes of k, a state $s \in S$, the max prefix, defined as mp_s, is the longest element in $\mathbb{P}^k$ such that $\exists t \in \mathbb{T}_s^e,\ mp_s \prec t$. The common prefix, defined as cp_s, is the longest element in $\mathbb{P}^k$ such that $\forall t \in \mathbb{T}_s^e,\ cp_s \prec t$.*

In a similar way we assign to each state of the LTS the suffixes of the sequence of inevitable actions k that will be completed starting from s. We introduce the notions of *max suffix* and *common suffix*, w.r.t. k. The max suffix is the longest suffix of the k sequence among the suffixes of traces that start from a given state. The common suffix is the longest suffix of the k sequence that is common to all the suffixes of traces that start from a given state. We define $\mathbb{T}_s^o$ as the set of all the suffixes of traces that start from s and $\mathbb{S}^k$ as the set of all the suffixes of k.

Definition 8. *(Max and Common Suffix) Given an LTS $M = (S, s^0, \Sigma, T)$, a sequence of inevitable actions k, the set $\mathbb{S}^k$ of all the suffixes of k, a state $s \in S$, the max suffix, defined as ms_s, is the longest element in $\mathbb{S}^k$ such that $\exists t \in \mathbb{T}_s^o,\ ms_s \prec t$. The common suffix, defined as cs_s, is the longest element in $\mathbb{S}^k$ such that $\forall t \in \mathbb{T}_s^o,\ cs_s \prec t$.*

The example given in Fig. 1 shows the max/common prefixes and suffixes calculated on each state of an LTS for a given sequence of inevitable actions $k = A \cdot Y$. Let us take a look at state 8: the *cp* value shows that the action A exists in every prefix of k produced by prefixes of traces that end in state 8. Conversely, the *cs* value in state 8 is empty while the *ms* value is $A \cdot Y$, meaning that the suffix $A \cdot Y$ is not contained in every suffix of traces that starts in state 8. As a matter of fact, we can see that the only suffix of traces that respects the k sequence is the one that begins with the transition $9 \xrightarrow{C} 10 \in T$.

In some cases inevitable execution properties might not be satisfied because of loops in which the execution of the system can remain infinitely. Our notion of

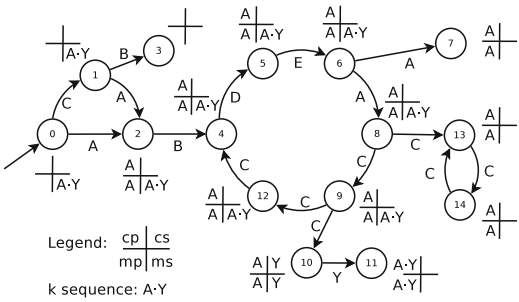


Fig. 1. Max/common prefixes and suffixes

suffix allows us to discover such loops and understand whether they prevent the satisfaction of the property. One of these loops is present in the example in Fig. 1 and is composed of states 4, 5, 6, 8, 9 and 12. These loops are treated in the next section by extracting the *Strongly Connected Components (SCCs)* [17] from the LTS. The LTS with the max/common prefixes (suffixes, resp.) computed for each state is called *enriched LTS*.

Definition 9. *(Enriched LTS) Given an LTS* $M = (S, s^0, \Sigma, T)$ *and a sequence of inevitable actions* k*, the enriched LTS is a tuple* $M_E^k = (S_E, s_E^0, \Sigma_E, T_E)$ *such that each state* $s_E \in S_E$ *is a tuple* $s_E = (s, mp_s, cp_s, ms_s, cs_s)$*, where* $s \in S$*,* $mp_s, cp_s \in \mathbb{P}_s^k$*,* $ms_s, cs_s \in \mathbb{S}_s^k$*;* $s_E^0 = (s^0, mp_{s^0}, cp_{s^0}, ms_{s^0}, cs_{s^0})$*;* $\Sigma_E = \Sigma$*;* $T_E \subseteq S_E \times \Sigma_E \times S_E$*, where* $\forall s \xrightarrow{l} s' \in T$*,* $(s, mp_s, cp_s, ms_s, cs_s) \xrightarrow{l} (s', mp_{s'}, cp_{s'}, ms_{s'}, cs_{s'}) \in T_E$*.*

3.2 Prefixes and Suffixes Calculation

This section presents the computation of the prefixes and suffixes defined in Sect. 3.1. In order to handle cycles in an LTS we use the notion of SCC, that is a partition of an LTS where every state is reachable from any other state. Note that every SCC in an LTS is also a Maximally Strongly Connected Component, since an SCC cannot be subsumed by a larger SCC by definition. To detect all the SCCs in an LTS we use the Tarjan's SCCs algorithm [17]. Given an LTS $M = (S, s^0, \Sigma, T)$, the algorithm allows the detection of all the SCCs in linear time, with a cost of $O(|S| + |T|)$. Given a sequence of inevitable actions k, our approach considers each SCC of the LTS, and computes the max/common prefixes and suffixes for every state in the SCC. Note that we start computing prefixes for states in a given SCC only when all its predecessors SCCs have been computed (in the case of suffixes we first compute all the successors).

We now introduce some notions related to the SCCs that we will use throughout the whole section. Given an LTS $M = (S, s^0, \Sigma, T)$ and an SCC G in M, where the sets of states and transitions in G are defined as S_G and T_G, respectively, we denote as $S_G^e \subseteq S_G$ the set of *initial states* of G, such that, given a transition $s \xrightarrow{l} s' \in T$, the state $s \notin S_G$ and $s' \in S_G^e$. The transition $s \xrightarrow{l} s' \in T$ is defined as *incoming transition* and the set of incoming transitions is written as T_G^e. Similarly, we denote as $S_G^o \subseteq S_G$ the set of *outgoing states* such that, given a transition $s \xrightarrow{l} s' \in T$, the state $s \in S_G^o$ and $s' \notin S_G$. The transition $s \xrightarrow{l} s' \in T$ is defined as an *outgoing transition* and the set of outgoing transitions is written as T_G^o. We denote as $\mathbb{G}_\mathbb{M}$ the *component graph* [7] of an LTS M where the states are given by SCCs of M. The SCC containing the initial state s_0 of the LTS M does not have any predecessors and it is defined as G^0. By definition, since all cycles are contained in SCCs, $\mathbb{G}_\mathbb{M}$ is a directed acyclic graph. The rest of this section presents the computation of the max prefix (suffix, resp.) and of the common prefix (suffix, resp.) for states of an SCC.

Max Prefix Calculation. The max prefix inside an SCC is computed by first extracting the longest max prefix among the incoming states of the SCC. Second,

the incoming max prefix is extended with actions contained inside the SCC to produce the longest (possible) prefix of k. Note that the max prefix is the same for all the states of an SCC. The cost of the computation for an SCC G is $O(|T_G^e| + |T_G| + |k|)$, since we first have to explore all the incoming transitions to compute the initial max prefix, and second we have to collect all the actions in the SCC that are also present in k. Let us consider the SCC composed of states 1, 2 and 3 in Fig. 2 (note that SCCs in states 0, 4 and 5 are trivial). Given $k = A \cdot B \cdot C$, the initial max prefix for the SCC is A, since the transition from state 0 to state 1 is the only incoming transition and it contains the first action of the k sequence. One can notice that by looping inside the SCC it is possible to complete the k sequence, since the SCC contains also actions B and C. Consequently, the max prefix in each state of the SCC (states 1, 2 and 3) is equivalent to the k sequence.

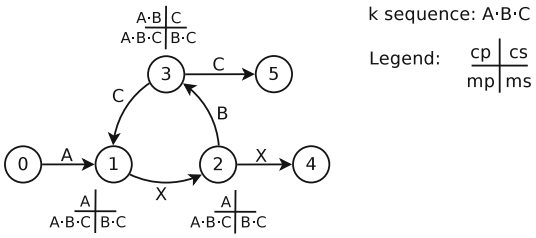


Fig. 2. Prefix and suffix calculation on an SCC

Max Suffix Calculation. The max suffix is computed similarly to the max prefix, by considering suffixes of successors instead of prefixes of predecessors. In the example in Fig. 2 the max suffix for every state in the SCC is $B \cdot C$, since they are the only two actions contained in the SCC that also exist in k.

Common Prefix Calculation. We describe here the computation of the common prefix for each state in an SCC. The pseudo-code of this procedure is detailed in Algorithm 1. The algorithm is divided into two main steps: initialisation and internal transitions computation.

Initialisation Step. Given an SCC G the algorithm initialises the common prefix of states in G to k (Line 3). There are two exceptions to this rule. First, the initial state of the LTS s^0 is initialised to the empty sequence since it has no predecessors. Second, if s is an initial state of G, cp_s is initialised with the common prefixes of its incoming transitions (Line 7). Let us take a look at the example in Fig. 2. The initialisation step assigns $cp = A \cdot B \cdot C$ to states 2 and 3, while it assigns $cp = A$ to state 1, which is the only initial state of the SCC.

Internal Transitions Computation Step. After the initialisation step, there may still be paths within G that can produce a prefix smaller than the ones in initial states. This step deals with internal transitions to detect smaller prefixes. First of all, we use Q (Line 8) as sorted set to order by increasing common prefix size

the states in S_G. When modifying the common prefix of a state s, UpdatePosition(Q, s) updates the position of s in Q. The loop of Line 9 iterates on Q, removing the first element s at each iteration. The common prefix of all successors s' of s within G is updated using a function LCP, which computes the longest common prefix between two sequences of actions. When producing a smaller prefix, the position of s' in Q is updated. Let us consider again the SCC in Fig. 2. The internal transitions computation step corrects the values of cp in states 2 and 3, assigning respectively $cp = A$ and $cp = A \cdot B$. The value of cp in state 1 remains the same, since it was already the lower one among all the states of the SCC.

Correctness and Complexity. The common prefix algorithm behaves similarly to the Dijkstra's algorithm that deals with the single-source shortest-paths problem in weighted directed graphs (in particular, to the implementation which uses a Fibonacci heap as priority queue). The complexity of the algorithm is $O(|T^e_G| + |T_G| + |S_G| \log |S_G|)$ because the while loop performs exactly $|S_G|$ iterations and the cost of inserting an element to Q and updating its position is $O(\log |Q|)$.

Algorithm 1. Common Prefix Computation

```
1:  procedure CommonPrefix(G, k)
2:      for all s ∈ S_G do
3:          cp_s ← k
4:          if s = s^0 then cp_s ← ∅
5:          else if s ∈ S^e_G then
6:              for all s' --l--> s ∈ ExtractIncomingTrans(s) do
7:                  if s' ∉ S_G then cp_s ← LCP(cp_s, cp_s' · l)
8:      Q ← S_G
9:      while Q ≠ ∅ do
10:         s ← PopFirst(Q)
11:         for all s --l--> s' ∈ T_G do
12:             t ← LCP(cp_s · l, cp_s')
13:             if |t| < |cp_s'| then cp_s' ← t ; UpdatePosition(Q, s')
```

Common Suffix Calculation. The common suffix calculation is similar to the prefix case, but it differs in the initialisation step. In the suffix case the execution may loop into the current SCC and never go through an outgoing transition, and a state may thus have a smaller common suffix than all states from its successor SCCs. This initialisation step is presented in Algorithm 2. In the case of a final state, the suffix is empty (Line 3). Otherwise, the common suffix is initialised to the smallest suffix of k traversed by a loop from s to itself (Line 4). In the absence of loops (SCC with single state and no self-loop), MinSuffixLoop returns k. The remainder of the computation is similar to Algorithm 1, using a function LCS, which computes the longest common suffix, instead of LCP. Searching the smallest-suffix loop for each state is done by iteratively removing labels from k and looking for isolated vertices. Hence, the overall cost of the computation is

$O(|T_G^o| + |k| \times (|T_G| + |S_G|) + |S_G| \log |S_G|)$. Let us consider the example in Fig. 2. The initialisation step assigns $cs = B \cdot C$ to state 1, which is the smallest suffix of k that can be produced inside the SCC starting from state 1. It then assigns the empty sequence and $cs = C$ to states 2 and 3, since they are outgoing states. The algorithm will later update the value of cs in state 1 to the empty sequence with the internal transitions computation step.

Algorithm 2. Common Suffix Computation (Initialisation Step)

```
1: procedure CommonSuffixInit(G, k)
2:     for all s ∈ S_G do
3:         if ∄ s --l--> s' then cs_s ← ∅
4:         else cs_s ← MinSuffixLoop(s, k, G)
5:             if s ∈ S_G^o then
6:                 for all s --l--> s' ∈ ExtractOutgoingTrans(s) do
7:                     if s' ∉ S_G then cs_s ← LCS(cs_s, l · cs_s')
```

Order of Calculation. So far, we have considered the computation of prefixes and suffixes for the states of an SCC. However, evaluating prefixes requires that the prefixes of all predecessor states of an SCC are correct (successors states in case of suffixes). It is thus important to execute our approach on SCCs in an appropriate order. Since by definition there are no cycles in $\mathbb{G}_{\mathbb{M}}$, we can define the *depth* of an SCC as 0 for the SCC that contains s_0, and 1 plus the maximum depth of predecessor SCCs otherwise. Our approach computes prefixes in SCCs by increasing depth, and suffixes by decreasing depth, ensuring the presence of the necessary information. Given the costs of computing prefixes and suffixes in each SCC, the total cost of the calculation of the enriched LTS is $O(|k| \times (|T| + |S|) + \sum_{G \in \mathbb{G}_{\mathbb{M}}} (|S_G| \log |S_G|))$.

3.3 Neighbourhoods

The enriched LTS with max/common prefixes and suffixes can now be used to characterise its transitions. A transition is typed as *correct* if it always leads to a correct part of the model, as *incorrect* if it always leads to an incorrect part of the model, as *neutral* if none of the previous cases apply.

More specifically, a *correct transition* leads to a portion of the LTS where the sequence of actions k is always respected. To state whether a transition is a correct one we compute the sum of the length of cp in the source state and of cs in its destination state. Note that we also have to take into account the label of the transition in this sum, since the concatenation of cp with the label may produce a valid prefix of k. If the sum is equal or higher than the size of the k sequence the transition is identified as correct.

Definition 10. *(Correct Transition) Given an enriched LTS* $M_E^k = (S_E, s_E^0, \Sigma_E, T_E)$*, two states* $s_E = (s, mp_s, cp_s, ms_s, cs_s) \in S_E$*,* $s'_E = (s', mp_{s'}, cp_{s'}, ms_{s'}, cs_{s'}) \in S_E$*, a correct transition is a transition* $s_E \xrightarrow{l} s'_E \in T_E$ *such that* $cp = cp_s \cdot l$ *if* $cp_s \cdot l$ *is a prefix of* k*,* $cp = cp_s$ *otherwise, and* $|cp| + |cs_{s'}| \geq |k|$*.*

On the contrary, an *incorrect transition* is a transition that leads to a portion of the LTS where the sequence of actions k is never respected. We take into account the sum of the length of mp in the source state and of ms in its destination state. If the sum is lower than the size of the k sequence the transition is classified as incorrect.

Definition 11. *(Incorrect Transition) Given an enriched LTS* $M_E^k = (S_E, s_E^0, \Sigma_E, T_E)$ *two states* $s_E = (s, mp_s, cp_s, ms_s, cs_s) \in S_E$, $s'_E = (s', mp_{s'}, cp_{s'}, ms_{s'}, cs_{s'}) \in S_E$, *an incorrect transition is a transition* $s_E \xrightarrow{l} s'_E \in T_E$ *such that* $mp = mp_s \cdot l$ *if* $mp_s \cdot l$ *is a prefix of* k, $mp = mp_s$ *otherwise, and* $|mp| + |ms_{s'}| < |k|$.

When a transition cannot be identified as correct nor as incorrect, it means that it is common to both correct and incorrect behaviours. Such transition is called a *neutral transition.*

The LTS where correct, incorrect and neutral transitions have been detected allows us to recognise *neighbourhoods* in states in which an incoming neutral transition is followed by a correct or an incorrect one. A neighbourhood represents a choice in the LTS between branching behaviours that directly affect the property satisfaction, and it consists of incoming and outgoing transitions of that state.

Definition 12. *(Neighbourhood) Given an LTS* $M_E^k = (S_E, s_E^0, \Sigma_E, T_E)$ *where transitions have been typed as correct, incorrect or neutral, a state* $s \in S_E$ *where* $\forall t = s' \xrightarrow{l} s \in T_E$, t *is a neutral transition and* $\exists t' = s \xrightarrow{l} s'' \in T_E$, t' *is a correct or an incorrect transition, the neighbourhood of state* s *is the set of transitions* $T_{nb} \subseteq T_E$ *such that for each* $t'' \in T_{nb}$, *either* $t'' = s' \xrightarrow{l} s \in T_E$ *or* $t'' = s \xrightarrow{l} s''' \in T_E$.

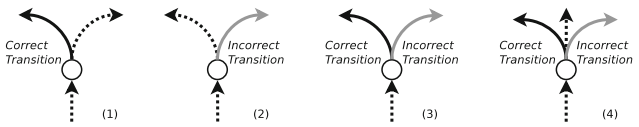


Fig. 3. The four types of neighbourhoods.

We can identify four different types of neighbourhoods by looking at their outgoing transitions (see Fig. 3 from left to right). The first type consists of neighbourhoods with at least one correct transition and no incorrect transitions, and highlights a choice that can lead to behaviours that always comply with the sequence of inevitable actions. The second type is represented by neighbourhoods with at least one incorrect transition but no correct transitions, and highlights a choice that can lead to behaviours that never comply with the sequence of inevitable actions. The third and the fourth types have both at least one correct

and one incorrect outgoing transition, and they differ because of the presence (or not) of one (or more) neutral outgoing transition(s).

In Fig. 4 we show the example described in Sect. 3.1 (in Fig. 1), where the transition detection has allowed to detect neighbourhoods (states coloured in grey). In particular state 9, where correct and neutral transitions are present, shows a neighbourhood of the first type, where a choice that will always satisfy the property is possible. On the contrary, states 1, 6 and 8 show neighbourhoods of the second type, where a choice that leads to an incorrect behaviour is possible.

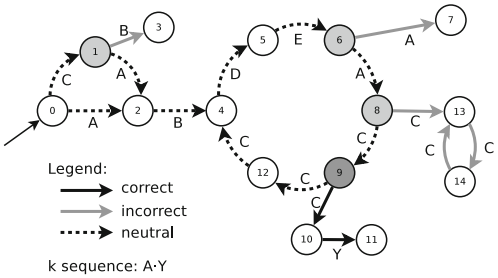


Fig. 4. Transitions classification and neighbourhoods

3.4 Simplification Techniques

The final step of our approach aims at simplifying the counterexample produced from the LTS and a given property. To do so we make a joint analysis of the counterexample and of the LTS enriched with neighbourhoods computed previously. This analysis can be used for obtaining different kinds of simplifications, such as: (i) an *abstracted counterexample*, that allows to remove from a counterexample actions that do not belong to neighbourhoods (and thus represent noise); (ii) a *shortest path to a neighbourhood*, which retrieves the shortest sequence of actions that leads to a neighbourhood; (iii) improved versions of (i) and (ii), where the user provides a pattern representing a sequence of non-contiguous actions, in order to allow the developer to focus on a specific part of the model; (iv) techniques focusing on a notion of *distance* to the bug in terms of neighbourhoods. For the sake of space, we focus on the abstracted counterexample in this paper.

Abstracted Counterexample. This technique abstracts a counterexample keeping only transitions that belong to neighbourhoods. The technique takes as input an LTS M where neighbourhoods have been identified, and a counterexample c, produced from M and from the inevitability property expressed by k. The procedure for the abstracted counterexample first identifies in c the states that belong to a neighbourhood in M. Second, it removes all the actions in c that do not represent incoming or outgoing transitions of neighbourhoods identified in the previous step. Figure 5 shows an example of a counterexample where two neighbourhoods, highlighted on the right side, have been detected and allow us to identify actions that are preserved in the abstracted counterexample.

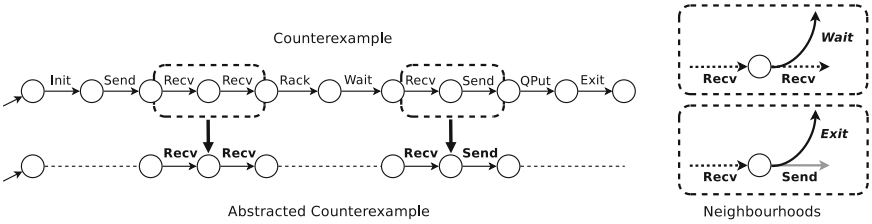


Fig. 5. Abstracted counterexample

4 Illustration on Case Studies

We implemented our approach in a Java tool, which consists of about 6000 lines of code. Compilers provided by the CADP toolbox [10] have been used to transform LNT [6] and LOTOS [5] specifications into LTS models, which are used as input format to our application. Our tool was applied to several examples in order to validate it. We present here two of them, showing the verification of both a simple and a nested inevitable execution property.

Sanitary Agency. We describe here the *Sanitary Agency* [16] case study, which models an agency that aims at supporting elderly citizens in receiving sanitary assistance from the public administration. The model is composed of four participants, depicted in Fig. 6: a bank to manage fees and payments; a cooperative to provide welfare services; a citizen to perform the service requests; a sanitary agency to manage citizens' requests and which also contribute to the payment. For illustration purposes, we defined a property with two nested inevitable executions. The property states that the treatment of a citizen request by the agency (represented by the `REQ_EM` action) should always take place, and should always be followed by the reception of a transport service by the citizen (represented by the `PROVT_REC` action).

Our tool identified five neighbourhoods in the model. We then applied the abstracted counterexample technique to the shortest counterexample, allowing to discover two neighbourhoods and consequently reducing the length of the counterexample from 15 actions to 4. The top side of Fig. 7 depicts the shortest counterexample while the bottom side depicts the corresponding neighbourhoods. The extracted actions are relevant since the neighbourhoods to which they belong precisely identify choices in the model that violate the property. In this case, the first neighbourhood shows that the first action in the property is not inevitable. The `REQ_EM` action can take place only after an `ACCEPTANCE_EM` action, but the neighbourhood exhibits an incorrect transition with the `REFUSAL_EM` action, revealing that the citizen request can be refused and thus preventing its treatment. The second neighbourhood shows that, even when the citizen request is treated by the agency, the system does not always satisfy the nested inevitable action, since it can also provide meal services. This is highlighted by

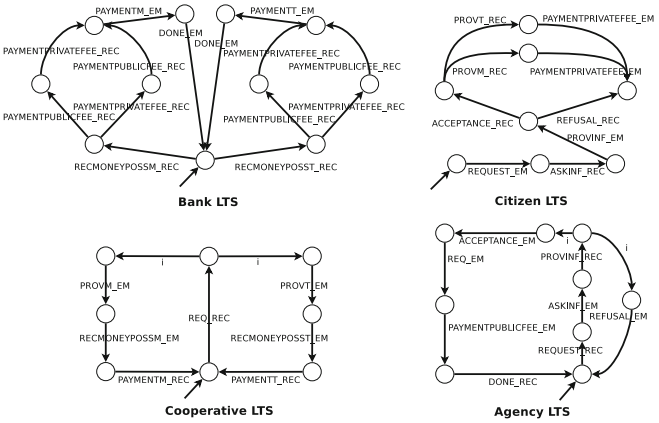


Fig. 6. Sanitary agency models

the choice between the correct transition with the `PROVT_EM` action (emission of a transport service) and the incorrect transition with the `PROVM_EM` action (emission of a meal service).

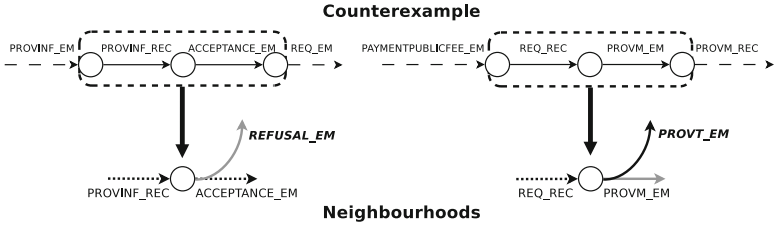


Fig. 7. Sanitary agency: shortest counterexample and neighbourhoods

Alternating Bit Protocol. We now discuss the *Alternating Bit Protocol* case study, which consists of a data link layer network protocol that allows the retransmission of lost or corrupted messages. The version of the protocol analysed here, available as CADP demo [12], is a variant without data values written in LOTOS. The model is composed of four processes: a transmitter process that acquires and sends a message; a receiver process that gets a message; medium1 and medium2 processes that represent transmission channels.

The demo is provided with an inevitable execution property that states that a `PUT` action will be eventually reached from the initial state. This property is not satisfied by the model because of the presence of loops in the specification that can lead to an infinite trace that never reaches a `PUT` action. More precisely, the problem is caused by an interaction between the receiver and the medium2 processes. When the transmitter process has not yet started the message treatment

(represented by the PUT action), the receiver might have to wait. In this case the receiver produces a TIMEOUT action, followed by the sending of an incorrect ack message (RACK1 action). If this ack message is lost by medium2 (LOSS action) and the receiver is still waiting, a loop might be produced until the transmitter starts the message treatment.

Our tool detects six neighbourhoods in the model, all with correct and neutral transitions. Figure 8 depicts a portion of the model with these neighbourhoods, which are located at states 0, 2, 5, 12, 13 and 23. In particular, neighbourhoods at states 5, 12, 13, 23 present choices that make the execution of the system remain infinitely inside the loops, preventing the satisfaction of the property by reaching the PUT action. This is highlighted by neutral transitions containing LOSS, RACK1 and TIMEOUT actions that repeat the cycle.

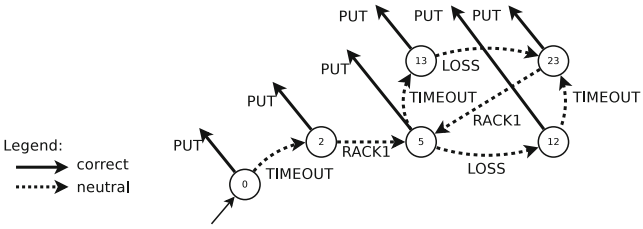


Fig. 8. Excerpt of the Alternating Bit protocol LTS model

5 Related Work

Causality analysis aims at relating causes and effects, which can help in debugging faults in (possibly concurrent) systems. This analysis relies on a notion of counterfactual reasoning, where alternative executions of the system are derived by assuming changes in the program. In [11], the authors present a general approach for causality analysis of system failures based on component specifications and observed component traces. In [3], the authors choose the Halpern and Pearl model to define causality checking in order to localise errors in hardware systems by analysing counterexample traces. Our approach is complementary to causality analysis since it helps to detect any kind of bugs and not only those involving causality.

In [4], sequential pattern mining is applied to execution traces for revealing unforeseen interleavings that may be a source of error, through the adoption of the well-known mining algorithm CloSpan. CloSpan is also adopted in [14], where the authors apply sequential pattern mining to traces of counterexamples, in order to reveal unforeseen interleavings that may be a source of error. However, reasoning on traces induces several issues. The handling of looping behaviours is non-trivial and may result in the generation of infinite traces or of an infinite number of traces. Coverage is another problem, since a high number of traces does not guarantee to produce all the relevant behaviours for analysis purposes.

As a result, we decided to work on the debugging of LTS models, which represent in a finite way all possible behaviours of the system.

In [13] the authors propose a method to interpret counterexamples traces from liveness properties by dividing them into fated and free segments. Fated segments represents inevitability w.r.t. the failure, pointing out progress towards the bug, while free segments highlight the possibility to avoid the bug. The proposed approach classifies states in different layers (representing distances from the bug) and produces a counterexample annotated with segments by exploring the model. Both our work and [13] aim at building an explanation from the counterexample. However, our method focuses on locating branching behaviours that affect the property satisfaction whereas their approach produces an enhanced counterexample where inevitable events (w.r.t. the bug) are highlighted.

Fault localisation for program debugging has been an active topic of research for many years in the software engineering community [18]. One of the main approaches in that line of work aims at localising faults using testing approaches. As an example, the approach presented in [15] relies on mutation testing to locate effectively the faulty statements. Experiments carried out in [15] reveal that mutation-based fault localisation is significantly more effective than current state-of-the-art fault localisation techniques. Note that this work applies on sequential C programs whereas we focus on formal models of concurrent programs.

We published in [2] an approach for counterexample analysis of safety property violation. [2] describes a preliminary version of neighbourhood and of the counterexample abstraction. The algorithmic solution using prefix/suffix annotations presented in Sect. 3 as well as the tool support and experiments presented in Sect. 4 are entirely new.

6 Conclusion

In this paper, we have proposed a method for improving the comprehension of counterexamples returned by a model checker when an inevitability property is not satisfied on a given behavioural model. To do so, we have first defined an algorithm to enrich the LTS that represents the model of the system with notions of prefixes and suffixes, which express parts of the sequence of inevitable actions. Second, we have provided a method to extract relevant portions of the LTS, called neighbourhoods, which highlight choices between a correct and an incorrect behaviour. Third, we have proposed a set of simplification techniques to extract relevant information that explains the cause of the bug, exploiting the notion of neighbourhood. All the steps of our approach are automated by a tool we implemented. The resulting simplified counterexample gives an improved explanation of the bug, as we have shown on experiments we carried out on real-world examples.

Acknowledgements. We are grateful to Radu Mateescu for his valuable inputs on liveness properties.

References

1. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
2. Barbon, G., Leroy, V., Salaün, G.: Debugging of concurrent systems using counterexample analysis. In: Dastani, M., Sirjani, M. (eds.) FSEN 2017. LNCS, vol. 10522, pp. 20–34. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68972-2_2
3. Beer, A., Heidinger, S., Kühne, U., Leitner-Fischer, F., Leue, S.: Symbolic causality checking using bounded model checking. In: Fischer, B., Geldenhuys, J. (eds.) SPIN 2015. LNCS, vol. 9232, pp. 203–221. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23404-5_14
4. Befrouei, M.T., Wang, C., Weissenbacher, G.: Abstraction and mining of traces to explain concurrency bugs. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 162–177. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_14
5. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Comput. Netw. **14**, 25–59 (1987)
6. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., Lang, F., McKinty, C., Powazny, V., Serwe, W., Smeding, G.: Reference Manual of the LNT to LOTOS Translator (Version 6.7). INRIA/VASY and INRIA/CONVECS, 153 pages (2018)
7. Clarke, E.M., Jha, Y., Lu, S., Veith, H.: Tree-like counterexamples in model checking. In: Proceedings of LICS 2002, pp. 19–29. IEEE Computer Society (2002)
8. De Nicola, R., Vaandrager, F.: Action versus state based logics for transition systems. In: Guessarian, I. (ed.) LITP 1990. LNCS, vol. 469, pp. 407–419. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-53479-2_17
9. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of ICSE 1999, pp. 411–420. ACM (1999)
10. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. STTT **15**(2), 89–107 (2013)
11. Gößler, G., Le Métayer, D.: A general trace-based framework of logical causality. In: Fiadeiro, J.L., Liu, Z., Xue, J. (eds.) FACS 2013. LNCS, vol. 8348, pp. 157–173. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07602-7_11
12. Inria CONVECS Team: CADP Demo 01: Alternating Bit Protocol. http://cadp.inria.fr/demos.html
13. Jin, H.S., Ravi, K., Somenzi, F.: Fate and FreeWill in error traces. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 445–459. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_31
14. Leue, S., Befrouei, M.T.: Mining sequential patterns to explain concurrent counterexamples. In: Bartocci, E., Ramakrishnan, C.R. (eds.) SPIN 2013. LNCS, vol. 7976, pp. 264–281. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39176-7_17
15. Papadakis, M., Traon, Y.L.: Effective fault localization via mutation analysis: a selective mutation approach. In: Proceedings of SAC 2014, pp. 1293–1300. ACM (2014)
16. Salaün, G., Bordeaux, L., Schaerf, M.: Describing and reasoning on web services using process algebra. In: Proceedings of ICWS 2004, pp. 43–50. IEEE Computer Society (2004)
17. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
18. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Trans. Softw. Eng. **42**(8), 707–740 (2016)

Online Enumeration of All Minimal Inductive Validity Cores

Jaroslav Bendík[1(✉)], Elaheh Ghassabani[2], Michael Whalen[2], and Ivana Černá[1]

[1] Faculty of Informatics, Masaryk University, Brno, Czech Republic
{xbendik,cerna}@fi.muni.cz
[2] Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA
{ghass013,mwwhalen}@umn.edu

Abstract. Symbolic model checkers can construct proofs of safety properties over complex models, but when a proof succeeds, the results do not generally provide much insight to the user. Minimal Inductive Validity Cores (MIVCs) trace a property to a minimal set of model elements necessary for constructing a proof, and can help to explain why a property is true of a model. In addition, the traceability information provided by MIVCs can be used to perform a variety of engineering analysis such as coverage analysis, robustness analysis, and vacuity detection. The more MIVCs are identified, the more precisely such analyses can be performed. Nevertheless, a full enumeration of all MIVCs is in general intractable due to the large number of possible model element sets. The bottleneck of existing algorithms is that they are not guaranteed to emit minimal IVCs until the end of the computation, so returned results are not known to be minimal until all solutions are produced.
In this paper, we propose an algorithm that identifies MIVCs in an *online* manner (i.e., one by one) and can be terminated at any time. We benchmark our new algorithm against existing algorithms on a variety of examples, and demonstrate that our algorithm not only is better in intractable cases but also completes the enumeration of MIVCs faster than competing algorithms in many tractable cases.

Keywords: Inductive validity cores · SMT-based model checking
Inductive proofs · Traceability · Proof cores

1 Introduction

Symbolic model checking using induction-based techniques such as IC3/PDR [6], k-induction [21], and k-liveness [5] can be used to determine whether properties hold of complex finite or infinite-state systems. Such tools are popular both because they are highly automated (often requiring no user interaction other than the specification of the model and desired properties), and also because, in the event of a violation, the tool provides a counterexample demonstrating a situation in which the property fails to hold. These counterexamples can be used

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 189–204, 2018.
https://doi.org/10.1007/978-3-319-92970-5_12

both to illustrate subtle errors in complex hardware and software designs [18,19] and to support automated test case generation [23,24].

If a property is proved, however, most model checking tools do not provide additional information. This can lead to situations in which developers have an unwarranted level of confidence in the behavior of the system. Issues such as vacuity [14], incorrect environmental assumptions [22], and errors either in English language requirements or formalization can all lead to failures of "proved" systems. Thus, even if proofs are established, one must approach verification with skepticism.

Recently, *proof cores*[1] have been proposed as a mechanism to determine which elements of a model are used when constructing a proof. This idea is formalized by Ghassabani et al. for inductive model checkers [8] as *Inductive Validity Cores* (IVCs). IVCs offer proof explanation as to why a property is satisfied by a model in a formal and human-understandable way. The idea lifts UNSAT cores [25] to the level of sequential model checking algorithms using induction. Informally, if a model is viewed as a conjunction of constraints, a minimal IVC (MIVC) is a set of constraints that is sufficient to construct a proof such that if any constraint is removed, the property is no longer valid. Depending on the model and property to be analyzed, there are many possible MIVCs, and there is often substantial diversity between the IVCs used for proof. In previous work [8–10,20] we have explored several different uses of IVCs, including:

Traceability: Inductive validity cores can provide accurate traceability matrices with no user effort. Given multiple IVCs, *rich traceability* matrices [20] can be automatically constructed that provide additional insight about *required* vs. *optional* design elements.

Vacuity Detection: Syntactic vacuity detection (checking whether all subformulae within a property are necessary for its validity) has been well studied [14]. IVCs allow a generalized notion of vacuity that can indicate weak or mis-specified properties even when a property is syntactically non-vacuous.

Coverage Analysis: Coverage analysis provides a metric as to whether a set of properties is adequate for the model. Several different notions of coverage have been proposed [4,13], but these tend to be very expensive to compute. IVCs provide an inexpensive coverage metric by determining the percentage of model atoms necessary for proofs of all properties.

Impact Analysis: Given a single (or for more accurate results, all) MIVC, it is possible to determine which requirements may be falsified by changes to the model. This analysis allows for selective regression verification of tests and proofs: if there are alternate proof paths that do not require the modified portions of the model, then the requirement does not need to be re-verified.

Design Optimization: A practical way of calculating all MIVCs allows synthesis tools to find a minimum set of design elements (optimal implementation) for a certain behavior. Such optimizations can be performed at different levels of synthesis.

[1] https://www.cadence.com/.

To be useful for these tasks, the generation process must be efficient and the generated IVCs must be accurate and precise (that is, sound and minimal). In previous work, we have developed an efficient *offline* algorithm [9] for finding all minimal IVCs based on the MARCO algorithm for MUSes [15]. The algorithm is considered *offline* because it is not until all IVCs have been computed that one knows whether the solutions computed are, in fact, minimal. In cases in which models contain many IVCs, this approach can be impractically expensive or simply not terminate.

In this paper, we propose a novel *online* algorithm for MIVC enumeration. With this algorithm, solutions are produced incrementally, and each solution produced is guaranteed to be minimal. Therefore, the algorithm produces at least some MIVCs even in the case of models for which a complete MIVC enumeration is intractable. Moreover, the proposed algorithm is often more efficient then the baseline MARCO also in the case of tractable models. We demonstrate this via an experimental evaluation.

The rest of the paper is organized as follows. In Sect. 2 we define all the necessary notions. Section 3 summarizes the existing techniques. In Sect. 4 we present our novel algorithm. Section 5 provides an example execution of our algorithm. Finally, Sects. 4.6 and 6 cover implementation details and present experimental results.

2 Preliminaries

A transition system (I, T) over a state space S consists of an initial state predicate $I : S \to \mathit{bool}$ and a transition step predicate $T : S \times S \to \mathit{bool}$. The notion of reachability for (I, T) is defined as the smallest predicate $R : S \to \mathit{bool}$ satisfying the following formulae:

$$\forall s \in S : I(s) \Rightarrow R(s)$$
$$\forall s, s' \in S : R(s) \wedge T(s, s') \Rightarrow R(s')$$

A safety property $P : S \to \mathit{bool}$ holds on a transition system (I, T) iff it holds on all reachable states, i.e., $\forall s \in S : R(s) \Rightarrow P(s)$. We denote this by $(I, T) \vdash P$. We assume the transition step predicate T is equivalent to a conjunction of transition step predicates $T_1, \ldots, T_n$, called top level conjuncts. In such case, T can be identified with the set of its top level conjuncts $\{T_1, \ldots, T_n\}$. By further abuse of notation, we write $T \setminus \{T_i\}$ to denote removal of top level conjunct T_i from T, and $T \cup \{T_j\}$ to denote addition of top level conjunct T_j to T.

Definition 1. *A set of conjuncts $U \subseteq T$ is an* Inductive Validity Core (IVC) *for $(I, T) \vdash P$ iff $(I, U) \vdash P$. Moreover, U is a* Minimal IVC (MIVC) *for $(I, T) \vdash P$ iff $(I, U) \vdash P$ and $\forall T_i \in U : (I, U \setminus \{T_i\}) \nvdash P$.*

Note, that the minimality (and dually maximality) in this work is with respect to the set inclusion and not wrt cardinality as e.g. in the MaxSAT problem. There can be multiple MIVCs with different cardinalities. For an illustration of the concepts on a particular transition system, please refer e.g. to the Altitude Switch example [9].

3 Existing Techniques

Consider first a naive enumeration algorithm that explicitly checks each subset of T for being an IVC and then finds the minimal IVCs using subset inclusion relation. The main disadvantage of this approach is the large number of checks since there are exponentially many subsets of T. We briefly describe existing techniques that can be used to find all MIVCs while checking only a a small portion of subsets of T for being IVCs. Most of the techniques were inspired by the MUS enumeration techniques [2,3,16] proposed in the area of constraint processing and applied by Ghassabani et al. [8,9].

Definition 2 (Inadequacy). *A set of conjuncts $U \subseteq T$ is an* inadequate *set for $(I,T) \vdash P$ iff $(I,U) \nvdash P$. Especially, $U \subseteq T$ is a* Maximal Inadequate Set (MIS) *for $(I,T) \vdash P$ iff U is inadequate and $\forall T_i \in (T \setminus U) : (I, U \cup \{T_i\}) \vdash P$.*

Inadequate sets are duals to inductive validity cores. Each $U \subseteq T$ is either inadequate set or an inductive validity core. In order to unify the notation, we use notation *inadequate* and *adequate*. Note that especially minimal inductive validity cores can be thus called minimal adequate sets.

The first property used to improve the naive enumeration algorithm is the *monotonicity* of adequacy with respect to the subset inclusion.

Lemma 1 (Monotonicity). *If a set of conjuncts $U \subseteq T$ is an adequate set for $(I,T) \vdash P$ then all its supersets are adequate for $(I,T) \vdash P$ as well:*

$$\forall U_1 \subseteq U_2 \subseteq T : (I,U_1) \vdash P \Rightarrow (I,U_2) \vdash P.$$

Symmetrically, if $U \subseteq T$ is an inadequate set for $(I,T) \vdash P$ then all its subsets are inadequate for $(I,T) \vdash P$ as well:

$$\forall U_1 \subseteq U_2 \subseteq T : (I,U_2) \nvdash P \Rightarrow (I,U_1) \nvdash P.$$

Proof. If $U_1 \subseteq U_2$ then reachable states of (I,U_2) form a subset of the reachable states of (I,U_1).

The monotonicity allows to determine status of multiple subsets of T while using only a single check for adequacy. For example, if a set $U \subseteq T$ is determined to be adequate, then all of its supersets are adequate and do not need to be explicitly checked. Let $Sup(U)$ and $Sub(U)$ denote the set of all supersets and subsets of U, respectively.

Every algorithm for computing MIVCs has to determine status (i.e. adequate or inadequate) of every subset of T. In order to distinguish the subsets whose status is already known from those whose status is not known yet, we denote the former subsets as *explored* subsets and the latter as *unexplored* subsets. Moreover, we distinguish *maximal* unexplored subsets:

- U_{max} is a *maximal unexplored subset* of T iff $U_{max} \subseteq T$, U_{max} is unexplored, and each of its proper supersets is explored.

Algorithm 1. A näive shrinking algorithm

input : $(I, U) \vdash P$
output: MIVC for $(I, U) \vdash P$
1 **for** $T_i \in U$ **do**
2 | **if** $(I, U \setminus \{T_i\}) \vdash P$ **then** $U \leftarrow U \setminus \{T_i\}$
3 **return** U

A straightforward way to find a (so far unexplored) MIVC of T is to find an unexplored adequate subset $U \subseteq T$ and turn U into an MIVC by a process called *shrinking*. A shrinking procedure iteratively attempts to remove elements from the set that is being shrunk, checking each new set for adequacy and keeping only changes that leave the set adequate. A näive example is shown in Algorithm 1.

Ghassabani et al. [9] proposed an *online* algorithm for MIVC enumeration which is based on the MUS enumeration algorithm MARCO [16]. The algorithm iteratively chooses maximal unexplored subsets and tests them for adequacy. Each maximal subset that is found to be adequate is then shrunk into a MIVC. This algorithm enumerates MIVCs in an online manner with a relatively steady rate of the enumeration. However, an evaluation of the algorithm shown that it is rather slow since the shrinking procedure can be extremely time consuming as each check for adequacy is in fact a model checking problem.

Therefore, Ghassabani et al. [9] proposed another algorithm which, instead of computing MIVCs in on online manner, rather computes only *approximately* minimal IVCs. In particular, it iteratively picks maximal unexplored subsets, checks them for adequacy, and turns the adequate subsets into approximately minimal IVCs using the approximation algorithm `IVC_UC` [8]. `IVC_UC` is able to identify IVCs which are often very close to actual MIVCs, yet cheap to compute. This enumeration algorithm computes approximately minimal IVCs, and identifies MIVCs *offline* at the very end of the computation. An experimental evaluation shows that the latter algorithm computes all MIVCs much faster than the algorithm based on shrinking. However, it does not enumerate MIVCs online and thus on some benchmarks may produce no MIVCs within a given time limit.

4 Grow-Shrink Algorithm

In this section, we propose a novel algorithm for online MIVC enumeration. The MIVCs are found using an improved shrinking procedure. Moreover, the algorithm uses a procedure *grow*, which is a dual of the shrinking procedure. The algorithm also maintains the set *Unexplored* of unexplored subsets.

We can effectively use the set *Unexplored* for speeding up the shrinking procedure. When testing the set $U \setminus \{T_i\}$ (see line 2 in Algorithm 1) we first check whether $U \setminus \{T_i\}$ is still unexplored. If $U \setminus \{T_i\}$ is already explored, then its status is already known and no test for adequacy is needed.

4.1 Shrink Procedure

In the following observation, we specify which explored subsets can be used to speed up the shrinking procedure.

Algorithm 2. Approximate grow

input : $(I, T) \vdash P$
input : inadequate $U \subset T$ for $(I, T) \vdash P$
input : set *Unexplored* of unexplored subsets of T
output: approximately maximal inadequate set for $(I, T) \vdash P$

```
1 M ← a maximal M ∈ Unexplored such that M ⊇ U
2 while (I, M) ⊢ P do
3 |   M_IVC ← IVC_UC((I, M), P)          // gets approximately minimal IVC
4 |   T_i ← choose T_i ∈ (M_IVC \ U)
5 |   M ← M \ {T_i}
6 return M
```

Observation 1. *Let U_1, U_2 be subsets of T such that U_1 is explored, U_2 is unexplored, and $U_1 \subset U_2$. Then U_1 is inadequate for $(I, T) \vdash P$.*
Symmetrically, if U_1, U_2 are subsets of T such that U_2 is explored, U_1 is unexplored, and $U_1 \subset U_2$. Then U_2 is adequate for $(I, T) \vdash P$.

Proof. If U_1 is adequate, then all of its supersets are necessarily adequate. Thus, if U_1 is determined to be adequate, then not just U_1 but also all of its supersets become explored. Since U_1 is explored and U_2 is unexplored, then U_1 is necessarily an inadequate subset of T.

In other words, during the shrinking procedure, we are guaranteed that whenever we find an explored set, this set is inadequate. Thus, as a further optimization in our algorithm we try to identify as many inadequate sets as possible before starting the shrinking procedure. The search for inadequate sets is done with the help of the grow procedure.

4.2 Grow Procedure

Recall that if a set is determined to be inadequate then all of its subsets are necessarily also inadequate. Therefore, the larger is the set that is determined to be inadequate, the more inadequate sets are explored. To identify inadequate sets as quickly as possible we search for maximal inadequate sets (MISes).

In order to find a MIS, we can find an inadequate set $U \subset T$ and use a process called *grow* which turns U to a MIS for $(I, T) \vdash P$. The grow procedure iteratively attempts to add elements from $T \setminus U$ to U, checking each new set for adequacy and keeping only changes that leave the set inadequate. Same as in the case of shrink procedure, we can use the set *Explored* to avoid checking sets whose status is already known. However, such grow procedure might still perform too many checks for adequacy and thus be very inefficient.

Instead, we propose to use a different approach. Algorithm 2 shows a procedure that, given an inadequate set U for $(I, T) \vdash P$, finds an *approximately* maximal inadequate set. It first finds some maximal unexplored set M such that $M \supseteq U$ and checks it for adequacy. If M is inadequate, then it is necessarily

Algorithm 3. Solving algorithm

```
1 Function Solve(I,U,P):
2 |   res ← CheckAdq(I, U, P)
3 |   if res = UNKNOWN then
4 |   |   approximateWarning ← true                    // a global variable
5 |   return (res = ADEQUATE)
```

a MIS (this is a straightforward consequence of Observation 1) Otherwise, if M is adequate then it is iteratively reduced until an inadequate set is found.

In particular, whenever M is found to be adequate, the approximative algorithm `IVC_UC` by Ghassabani et al. [8] is used to find an approximately minimal IVC M_{IVC} of M. M_{IVC} succinctly explains M's adequacy. In order to turn M into an inadequate set, it is reduced by one element from $M_{IVC} \setminus U$ and checked for adequacy. If M is still adequate then the approximate growing procedure continues with a next iteration. Otherwise, if M is inadequate, the procedure finishes.

Proposition 1. *Given an unexplored inadequate set U for $(I, T) \vdash P$ and a set Unexplored of unexplored subsets of T, Algorithm 2 returns an* unexplored *inadequate subset M of T.*

Proof. Let us denote initial M as M_{init}. Since $M_{init} \supseteq U$ and M is recursively reduced only by elements that are not contained in U, then in every iteration holds that $U \subseteq M \subseteq M_{init}$. Since both U, M_{init} are unexplored, then M is necessarily also unexplored.

4.3 Solve Procedure

Determining whether a particular subset of elements $U \subset T$ can prove a property of interest P is as hard as model checking ([8], Theorem 1). Thus, in the general case, determining whether a set of model elements is an MIVC may not be possible for model checking problems that are in general undecidable, such as those involving infinite theories. We assume there is a function `CheckAdq` that checks whether or not P is provable for some (I, U). `CheckAdq` can return UNKNOWN (after a user-defined timeout) as well as ADEQUATE or INADEQUATE. For a given set U, if our implementation is unable to prove the property, we conservatively assume that the property is falsifiable and set a global warning flag *approximateWarning* to the user that the results produced may be approximate.

4.4 Complete Algorithm

In this section, we describe, how to combine the shrink and grow methods to form an efficient online MIVC enumeration algorithm. We call the algorithm Grow-Shrink algorithm. Since knowledge of (approximately) maximal inadequate subsets can be exploited to speed up the shrinking procedure, it might

Algorithm 4. The Grow-Shrink algorithm

```
1  Function Init((I, T) ⊢ P):
2      Unexplored ← 𝒫(T)                                   // a global variable
3      shrinkingQueue ← empty queue                        // a global variable
4      approximateWarning ← false                          // a global variable
5      FindMIVCs()
1  Function FindMIVCs():
2      while Unexplored ≠ ∅ do
3          U_max ← a maximal set ∈ Unexplored
4          if Solve(I, U_max, P) then
5              U_IVC ← IVC_UC((I, U_max), P)
6              Shrink(U_IVC)
7          else
8              Unexplored ← Unexplored \ Sub(U_max)
9          while shrinkingQueue is not empty do
10             U ← Dequeue(shrinkingQueue)
11             Shrink(U)
1  Function Shrink(U):
2      growingQueue ← empty queue
3      for T_i ∈ U do
4          if U \ {T_i} ∈ Unexplored then
5              if Solve(I, U \ {T_i}, P) then  U ← U \ {T_i}
6              else  Enqueue(growingQueue, U \ {T_i})
7      output U                                            // Output Minimal IVC
8      UpdateShrinkingQueue(U)
9      Unexplored ← Unexplored \ Sup(U)
10     while growingQueue is not empty do
11         V ← Dequeue(growingQueue)
12         Grow(V)
1  Function Grow(V):
2      M ← a maximal set ∈ Unexplored such that M ⊇ V
3      while Solve(I, M, P) do
4          M_IVC ← IVC_UC((I, M), P)
5          UpdateShrinkingQueue(M_IVC)
6          Enqueue(shrinkingQueue, M_IVC)
7          Unexplored ← Unexplored \ Sup(M_IVC)
8          T_i ← choose T_i ∈ (M_IVC \ V)
9          M ← M \ {T_i}
10     Unexplored ← Unexplored \ Sub(M)
1  Function UpdateShrinkingQueue(U):
2      for V ∈ shrinkingQueue do
3          if U ⊆ V then remove V from shrinkingQueue
```

be tempting to first find all MISes. However, this is in general intractable since there can be up to exponentially many MISes (w.r.t. the size of T). Instead, we propose to alternate both the shrinking and growing procedures. Note that during shrinking, we might determine some subsets to be inadequate. Such subsets can be subsequently used as *seeds* for growing. Dually, adequate subsets that are explored during growing can be later used as *seeds* for the shrinking procedure.

The pseudocode of our algorithm is shown in Algorithm 4. The computation of the algorithm starts with an initialisation procedure `Init` which creates a global variable *Unexplored* for maintaining the unexplored subsets and a global shrinking queue *shrinkingQueue* for storing seeds for the shrinking procedure. Then the main procedure `FindMIVCs` of our algorithm is called.

Procedure `FindMIVCs` works iteratively. In each iteration, the procedure picks a maximal unexplored subset U_{max} and checks it for adequacy. If U_{max} is inadequate, then U_{max} and all of its subsets are marked as explored. Otherwise, if U_{max} is adequate, then the algorithm `IVC_UC` [8] is used to reduce U_{max} into an approximately minimal IVC, and subsequently the procedure `Shrink` is used to shrink it into a MIVC.

Procedure `Shrink` works as described in Sect. 4.1. However, besides shrinking the given set into a MIVC, the procedure has also another purpose. Every inadequate set that is found during the shrinking is stored in a queue *growingQueue*. At the end of the procedure, all of these inadequate sets are grown into approximately maximal inadequate sets using the procedure `Grow`.

Procedure `Grow` turns a given inadequate set V into an approximately maximal inadequate set M as described in Sect. 4.2. The resultant set and all of its subsets are marked as explored. Moreover, every adequate set found during the growing is marked as explored and enqueued into *shrinkingQueue*. The queue *shrinkingQueue* is dequeued at the end of each iteration of the main procedure `FindMIVCs` and the sets that were stored in the queue are shrunk to MIVCs.

We need to ensure that each result of the shrinking procedure is a *fresh* MIVC, i.e. that each MIVC is produced only once. We shrink two kinds of inadequate sets in our algorithm: those that result from the inadequate maximal unexplored subsets, and those that are stored in *shrinkingQueue*. In the former case, we always shrunk an unexplored subset U_{IVC} which guarantees that the resultant MIVC U_{MIVC} is unexplored and thus fresh (if U_{MIVC} is already explored, then U_{IVC} would be necessarily also explored). However, in the latter case, all the sets stored in *shrinkingQueue* are already explored. To guarantee that shrinking of the sets from *shrinkingQueue* result only in fresh MIVCs, we maintain the following invariants of the queue:

(I1) For each already produced MIVC M holds that there is no U in the queue such that $M \subseteq U$.
(I2) There are no two U, V in the queue such that $U \subseteq V$.

To ensure that the invariants hold, we use the procedure `UpdateShrinkingQueue` which given an adequate set U removes from *shrinkingQueue* all supersets of U. We call the procedure every time a MIVC is found and every time a set is added to the queue.

Correctness: The algorithm produces only the MIVCs found by the shrinking procedure and all of them are *fresh*, i.e. produced only once. Only subsets whose status is known are removed from the set *Unexplored*, thus no MIVC is excluded from the computation. The algorithm terminates and all MIVCs are found since the size of *Unexplored* is reduced after every iteration.

4.5 Symbolic Representation of Unexplored Subsets

Since there are exponentially many subsets of T, it is intractable to represent the set *Unexplored* explicitly. Instead, we use a symbolic representation that is based on a well known isomorphism between finite power sets and Boolean algebras. We encode $T = \{T_1, T_2, \ldots, T_n\}$ by using a set of Boolean variables $X = \{x_1, x_2, \ldots, x_n\}$. Each valuation of X then corresponds to a subset of T. This allows us to represent the set of unexplored subsets *Unexplored* using a Boolean formula $f_{Unexplored}$ such that each model of $f_{Unexplored}$ corresponds to an element of *Unexplored*. The formula is maintained as follows:

- Initially, $f_{Unexplored} = \mathit{True}$ since all of $\mathcal{P}(T)$ are unexplored.
- To remove an adequate set $U \subseteq T$ and all its supersets from the set *Unexplored* we add to $f_{Unexplored}$ the clause $\bigvee_{i:T_i \in U} \neg x_i$.
- To remove an inadequate set $U \subseteq T$ and all its subsets from the set *Unexplored* we add to $f_{Unexplored}$ the clause $\bigvee_{i:T_i \notin U} x_i$.

In order to get an element of *Unexplored*, we ask a SAT solver for a model of $f_{Unexplored}$. In particular, to get a maximal unexplored subset, we ask a SAT solver for a *maximal model* of $f_{Unexplored}$. To get a maximal unexplored superset of $U \subseteq T$, we fix the truth assignment to the Boolean variables that correspond to elements in U to *True* and ask for a maximal model of $f_{Unexplored}$.

Example 1. Let us illustrate the symbolic representation on $T = \{T_1, T_2, T_3\}$. If all subsets of T are unexplored then $f_{Unexplored} = \mathit{True}$. If $\{T_1, T_3\}$ is classified as an MIVC and $\{T_1, T_2\}$ as a inadequate set, then $f_{Unexplored}$ is updated to $\mathit{True} \wedge (\neg x_1 \vee \neg x_3) \wedge (x_3)$.

4.6 Implementation

We have implemented[2] the Grow-Shrink algorithm in an industrial model checker called `JKind` [7], which verifies safety properties of infinite-state synchronous systems. It accepts Lustre programs [12] as input. The translation of Lustre into a symbolic transition system in `JKind` is straightforward and is similar to what is described in [11]. Verification is supported by multiple "proof engines" that execute in parallel, including K-induction, property directed reachability (PDR), and lemma generation. During verification, `JKind` emits SMT problems using the theories of linear integer and real arithmetic, and can use the `Z3`, `Yices`, `MathSAT`, `SMTInterpol`, and `CVC4` SMT solvers as back-ends. When a property

[2] https://github.com/jar-ben/jkind/tree/newalgorithm-shrink-tracking.

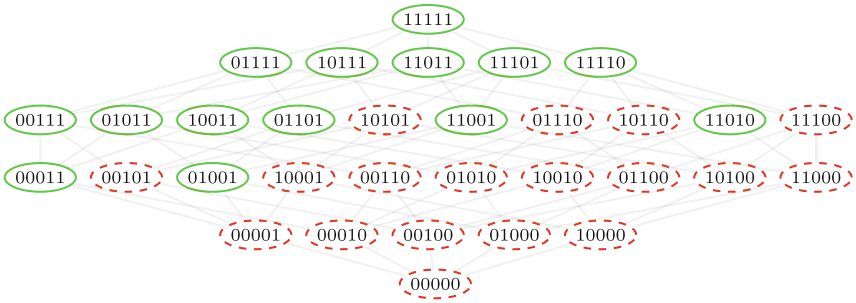


Fig. 1. The power set from the example execution of our algorithm. (Color figure online)

is proved and IVC generation is enabled, an additional parallel engine executes the `IVC_UC` algorithm [8] to generate an (approximately) minimal IVC. To implement our method, we have extended `JKind` with a new engine that implements Algorithm 4 on top of `Z3`.

5 Example Execution of the Grow-Shrink Algorithm

The following example explains the execution of our algorithm on a simple instance where the transition step predicate T is given as a conjunction of five sub-predicates $\{T_1, T_2, T_3, T_4, T_5\}$. We do not exactly state what are the predicates and what is the safety property of interest. Instead, Fig. 1 illustrates the power set of $\{T_1, T_2, T_3, T_4, T_5\}$ together with an information about adequacy of individual subsets. The subsets with solid green border are the adequate subsets, and the subsets with dashed red border are the inadequate ones. To save space, we encode subsets as bitvectors, for example the subset $\{T_1, T_2, T_4\}$ is written as 11010. There are three MIVCs in this example: 00011, 01001, and 11010.

We illustrate the first iteration of the main procedure `FindMIVCs` of our algorithm. Initially, all subsets are unexplored, i.e. $f_{Unexplored} = True$ and the queue *shrinkingQueue* is empty. The procedure starts by finding a maximal unexplored subset and checking it for adequacy. In our case, $U_{max} = 11111$ is the only maximal unexplored subset and it is determined to be adequate. Thus, the algorithm `IVC_UC` is used to compute an approximately minimal IVC $U_{IVC} = 01101$ which is then shrunk to a MIVC 01001.

During the shrinking, sets 00101, 01001, and 01000 are subsequently checked for adequacy and determined to be inadequate, adequate, and inadequate, respectively. The set 01001 is the resultant MIVC, thus the formula $f_{Unexplored}$ is updated to $f_{Unexplored} = True \wedge (\neg x_2 \vee \neg x_5)$. The other two sets, 00101 and 01000, are enqueued to the *growingQueue* and grown at the end of the procedure.

We first grow the set 00101. Initially, the procedure `Grow` picks $M = 10111$ as the maximal unexplored superset of 00101, and checks it for adequacy. It is adequate and thus, an approximately minimal IVC $M_{IVC} = 00011$ is computed,

enqueued to *shrinkingQueue*, and formula $f_{Unexplored}$ is updated to $f_{Unexplored} = True \wedge (\neg x_2 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_5)$. Then, M is (based on M_{IVC}) reduced to $M = 10101$ and checked for adequacy. It is found to be inadequate, thus formula $f_{Unexplored}$ is updated to $f_{Unexplored} = True \wedge (\neg x_2 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_2 \vee x_4)$, and the procedure terminates.

The growing of the set 01000 results into an approximately maximal inadequate subset 01110. Moreover, an approximately minimal IVC 11110 is found during the growing and enqueued into *shrinkingQueue*. The formula $f_{Unexplored}$ is updated to $f_{Unexplored} = True \wedge (\neg x_2 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_1 \vee x_5)$.

After the second grow, the procedure `Shrink` terminates and the main procedure `FindMIVCs` continues. The queue *shrinkingQueue* contains two sets: 00011, 11110, thus the procedure now shrinks them. During shrinking the set 00011, the algorithm would attempt to check the sets 00001 and 00010 for adequacy, however since both these are already explored, the set 00011 is identified to be a MIVC without performing any adequacy checks. The procedure `FindMIVCs` would now shrink also the set 11110, thus empty the queue *shrinkingQueue*, and continue with a next iteration.

6 Experiment

We are interested in examining the performance of algorithms to compute minimal IVCs. We examine **Grow-Shrink**, the algorithm presented in this paper, and the two state-of-the-art algorithms (briefly described in Sect. 3): **Offline MARCO**, the algorithm from [9], and **Online MARCO**, a variant of the algorithm from [9] that performs a shrink step prior to returning a solution to ensure minimality. We investigate the following research questions: (RQ1:) For the large models where the complete MIVC enumeration is intractable, how many MIVCs are found within the given time limit? (RQ2:) For the tractable models, i.e. models in which all MIVCs are found, how much time is required to complete the enumeration of MIVCs? Finally, we are interested in how many solver calls are necessary for the enumeration. Thus, we add (RQ3:) What is the (average) number of solver calls with result adequate/inadequate required by evaluated online algorithms to produce individual MIVCs?

Experimental Setup: We start from a benchmark suite that is a superset of the benchmarks used in [9]. This suite contains 660 models, and includes all models that yield a valid result (530 in total) from previous Lustre model checking papers [11,17] and 130 industrial models yielding valid results derived from an infusion pump system [19] and other sources [1,17]. As this paper is concerned with analysis problems involving multiple MIVCs, we include only models that had more than 4 MIVCs (46 models in total). To consider problems with many IVCs, we took those models and mutated them, constructing 20 mutants for each model. The mutants varied both in the number and in the size of individual MIVCs. We added the mutants that still yielded valid results and have

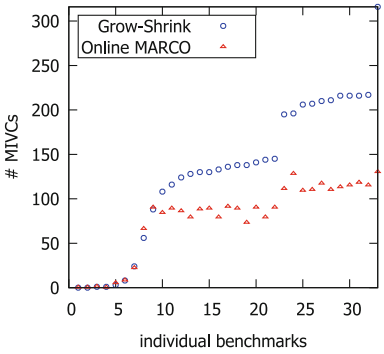


Fig. 2. Number of MIVCs produced by online algorithms.

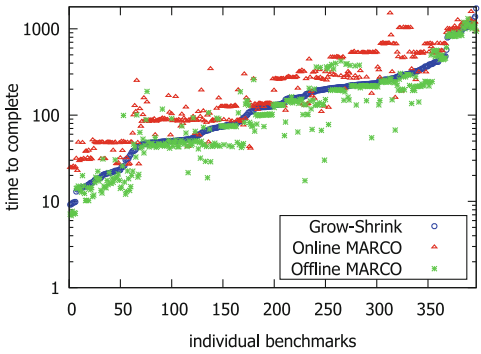


Fig. 3. Runtime for tractable benchmarks for all algorithms in a log scale.

more than 5 MIVCs (384 in total) back to the benchmark suite. Thus, the final suite contains 430 Lustre models. The original benchmarks and our augmented benchmark are available online[3].

For each test model, we configured `JKind` to use the `Z3` solver and the "fastest" mode of `JKind` (which involves running the k-induction and PDR engines in parallel and terminating when a solution is found). The experiments were run on a 3.50 GHz Intel(R) i5-4690 processor 16 GB memory machine running Linux with a 30 min timeout. All experimental data is available online[4].

6.1 Experimental Results

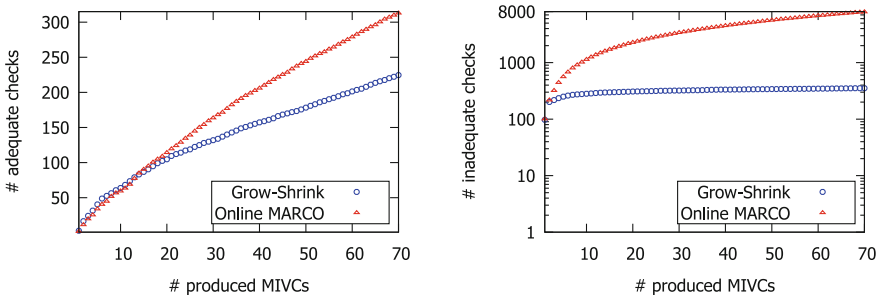
In this section, we examine the experimental results to address the research questions.

RQ1 and RQ2: Data related to the first two research questions is shown in Figs. 2 and 3. Figure 2 describes the number of MIVCs found be the two online algorithms in the intractable benchmarks, i.e. the benchmarks where the algorithms did not complete the computation within the time limit. There are 33 such benchmarks. Grow-Shrink substantially outperforms Online MARCO in the majority of the benchmarks, finding an average of 55% additional MIVCs.

Figure 3 describes the time for each algorithm needed to complete the computation in the case of 397 tractable benchmarks. Grow-Shrink is on average only 1.08 times slower than Offline MARCO, yet as previously discussed, has the advantage of returning guaranteed MIVCs, rather than approximate MIVCs. It is on average 1.50 times faster than Online MARCO.

RQ3: For RQ3, we examined the number of required calls to the solver per MIVC. For this question, we used the 33 models that contained a large number

[3] https://github.com/elaghs/benchmarks.
[4] https://github.com/jar-ben/online-mivc-enumeration.



(a) Checks with result "adequate". (b) Checks with result "inadequate".

Fig. 4. Average number of performed adequacy checks required to produce individual MIVCs. Note that Figure (b) is in a log scale.

of MIVCs (>70) in order to show the solver efficiency as the number of MIVCs increased. A point with coordinates (x, y) states that the algorithm needed to perform y solver calls (on average) in order to produce (find) the first x MIVCs. We grouped the calls in terms of the number of calls that returned *adequate* vs. *inadequate* results. It is evidenced by the results in Fig. 4, the new algorithm improves upon Online MARCO as the number of MIVCs becomes larger.

The improvement in the number of *inadequate* calls is due the novel shrinking and growing procedures. Each (approximately) maximal inadequate subset found by the growing procedure allows to save (up to exponentially) many inadequate calls during subsequent executions of the shrinking procedure. Indeed, the Grow-Shrink algorithm performed on average only 353 inadequate calls to output the first 70 MIVCs, whereas the online MARCO needed to perform 7775 calls to output the same number of MIVCs.

The improvement in the number of *adequate* calls is not so significant as in the case of inadequate calls. Yet, since the adequate calls are usually much more time consuming than inadequate ones, even a slight saving in the number of adequate calls might significantly speed up the whole computation. The Grow-Shrink algorithm saves adequate calls due to the usage of the shrinking queue and due to the invariants that are maintained by the queue. In particular, shall two comparable sets appear in the queue, only the smaller is left. Thus, the algorithm avoids shrinking of relatively large sets and saves some adequate calls.

7 Conclusion

We have presented an *online* algorithm, called *Grow-Shrink algorithm*, for computation of minimal Inductive Validity Cores (MIVCs). The new algorithm substantially outperforms previous approaches. As opposed to the *Offline MARCO* algorithm in [9], it is guaranteed to produce minimal IVCs. As opposed to a näive extension *Online MARCO*, the new algorithm is substantially faster and requires fewer solver calls as the number of MIVCs increases. We believe that

this new algorithm will substantially increase the applicability of software engineering tasks that require MIVCs. In the future, we hope to examine parallel computation of MIVCs using a variant of this algorithm to further increase scalability.

Acknowledgements. This work has been partially supported by the Czech Science Foundation grant No. 18-02177S.

References

1. Backes, J., Cofer, D., Miller, S., Whalen, M.W.: Requirements analysis of a quad-redundant flight control system. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 82–96. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_7
2. Bendík, J., Beneš, N., Barnat, J., Černá, I.: Finding boundary elements in ordered sets with application to safety and requirements analysis. In: De Nicola, R., Kühn, E. (eds.) SEFM 2016. LNCS, vol. 9763, pp. 121–136. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41591-8_9
3. Bendík, J., Benes, N., Cerná, I., Barnat, J.: Tunable online MUS/MSS enumeration. In: FSTTCS 2016, pp. 50:1–50:13 (2016)
4. Chockler, H., Kupferman, O., Vardi, M.Y.: Coverage metrics for formal verification. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 111–125. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39724-3_11
5. Claessen, K., Sörensson, N.: A liveness checking algorithm that counts. In: FMCAD, pp. 52–59. IEEE (2012)
6. Een, N., et al.: Efficient implementation of property directed reachability. In FMCAD 2011 (2011)
7. Gacek, A., Backes, J., Whalen, M., Wagner, M., Ghassabani, E.: The Jkind model checker (2017). arXiv preprint arXiv:1712.01222
8. Ghassabani, E., et al.: Efficient generation of inductive validity cores for safety properties. In: FSE 2016 (2016)
9. Ghassabani, E., Gacek, A., Whalen, M.W.: Efficient generation of all minimal inductive validity cores. In: FMCAD 2017 (2017)
10. Ghassabani, E., Gacek, A., Whalen, M.W., Heimdahl, M., Lucas, W.: Proof-based coverage metrics for formal verification. In: ASE 2017 (2017)
11. Hagen, G., Tinelli, C.: Scaling up the formal verification of lustre programs with SMT-based techniques. In: FMCAD 2008 (2008)
12. Halbwachs, N., et al.: The synchronous dataflow programming language Lustre. In: Proceedings of the IEEE (1991)
13. Kupferman, O., Li, W., Seshia, S.: A theory of mutations with applications to vacuity, coverage, and fault tolerance. In: FMCAD 2008, p. 25 (2008)
14. Kupferman, O., Vardi, M.Y.: Vacuity detection in temporal model checking. STTT **4**(2), 224–233 (2003)
15. Liffiton, M., et al.: Fast, flexible MUS enumeration. Constraints **21**(2), 223–250 (2016)
16. Liffiton, M.H., Previti, A., Malik, A., Marques-Silva, J.: Fast, flexible MUS enumeration. Constraints **21**(2), 223–250 (2016)
17. Mebsout, A., Tinelli, C.: Proof certificates for SMT-based model checkers for infinite-state systems. In: FMCAD 2016 (2016)

18. Miller, S.P., Whalen, M.W., Cofer, D.D.: Software model checking takes off. Commun. ACM **53**(2), 58–64 (2010)
19. Murugesan, A., et al.: Compositional verification of a medical device system. In: HILT 2013 (2013)
20. Murugesan, A., et al.: Complete traceability for requirements in satisfaction arguments. In: RE 2016 (RE@Next! Track) (2016)
21. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A., Johnson, S.D. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 127–144. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40922-X_8
22. Whalen, M., Cofer, D., Miller, S., Krogh, B.H., Storm, W.: Integration of formal analysis into a model-based software development process. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 68–84. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79707-4_7
23. Whalen, M., Gay, G., You, D., Heimdahl, M., Staats, M.: Observable modified condition/decision coverage. In: ICSE 2013. ACM (2013)
24. You, D., Rayadurgam, S., Whalen, M., Heimdahl, M.: Efficient observability-based test generation by dynamic symbolic execution. In: ISSRE 2015 (2015)
25. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable boolean formula. In: SAT 2003 (2003)

Prevent: A Predictive Run-Time Verification Framework Using Statistical Learning

Reza Babaee(✉), Arie Gurfinkel, and Sebastian Fischmeister

Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada
{rbabaeec,arie.gurfinkel,sebastian.fischmeister}@uwaterloo.ca

Abstract. Run-time Verification (RV) is an essential component of developing cyber-physical systems, where often the actual model of the system is infeasible to obtain or is not available. In the absence of a model, i.e., black-box systems, RV techniques evaluate a property on the execution path of the system and reach a verdict that the current state of the system satisfies or violates a given property. In this paper, we introduce *Prevent*, a predictive runtime verification framework, in which if a property is not currently satisfied, the monitor generates the probability based on the finite extensions of the execution path, that satisfy the specification property. We use Hidden Markov Model (HMM) to extend the partially observable paths of the system. The HMM is trained on a set of *iid* samples generated by the system. We then use reachability analysis to construct a lookup table that provides the probability that the extended path satisfies or violates the specification from the current state. The current state is estimated at run-time using Viterbi algorithm that gives the most probable state. We show an empirical evaluation of *Prevent* on a version of randomized dining philosopher and on the QNX Neutrino kernel traces collected from an autopilot software of a hexacopter.

1 Introduction

Run-time Verification (RV) [17] has become a crucial element in developing Cyber-Physical Systems (CPSs) [32,40,42]. In RV, a monitor checks the current execution, that is a finite prefix of an infinite path, against a given property, typically expressed in Linear Temporal Logic (LTL) [23], that represents a set of acceptable infinite paths. If any infinite extension of a prefix belongs (does not belong) to the set of infinite paths that satisfy the property, the monitor accepts (resp. rejects) the prefix. For example, $\varphi_{\mathsf{F}} : \Diamond e$ (resp. $\varphi_{\mathsf{G}} : \Box\neg e$) is satisfied (resp. not satisfied) on any infinite paths with the prefix $u_1 : \neg e,\neg e, e$. Whenever the monitor is not able to reach a verdict with the given prefix π because π can be extended to satisfy and to violate the property, the monitor outputs *unknown* [3]. For example, the prefix $u_2 : \neg e,\neg e$ can be extended to both a path that satisfies $\varphi_{\mathsf{F}} : \Diamond e$ (e.g., any extension of u_1) and a path that violates φ_{F} (e.g., $(\neg e)^{\omega}$).

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 205–220, 2018.
https://doi.org/10.1007/978-3-319-92970-5_13

The monitor is able to reach a verdict with a finite extension of the prefix, if the property is *monitorable* [12]. In this paper, we estimate the finite extensions of a prefix using a prediction model. The prediction model is trained from identically and independently distributed (*iid*) samples of the previous execution paths of the system. We use Hidden Markov Models [26] (HMMs) to realize a prediction model of the system with partially observable behaviour.

We focus on the properties that can be evaluated with regular extensions, that is, the extensions that are expressible by a Deterministic Finite Automaton (DFA). Depending on the given property, the extensions may specify the prefixes that satisfy the property (*good extensions*) or violate it (*bad extensions*). We use an upper-bound on the length of the estimated extensions. The monitor in our framework is the result of a bounded reachability analysis on the product of an HMM and a DFA. Using the product model, the monitor is able to predict a verdict, in terms of the probability of the extensions that satisfy or violate the property. To extend an execution path, the monitor needs to know the current state, which is estimated at run-time by Viterbi algorithm [38]. Viterbi algorithm generates the most likely state based on a given observation.

We implemented our approach as a proof-of-concept tool[1], called $\mathcal{P}revent$ (Predictive Runtime Verification Framework), and report on using it in two case studies: the original and a modified version of randomized dining philosophers algorithm, and the QNX Neutrino [24] kernel traces. In summary, we make the following contributions:

- introduce $\mathcal{P}revent$, a predictive runtime verification framework to detect satisfaction/violation of a property based on partial execution,
- methodology for constructing a prediction model, that is, the product of a trained HMM and the DFA specifying good and bad extensions,
- define the prediction error on a partial trace and evaluate the monitor performance using hypothesis testing,
- implement the runtime monitoring algorithm using Viterbi approximation,
- evaluate $\mathcal{P}revent$ with two case studies: a modified version of the randomized dining philosophers problem and the flight control of a hexacopter.

The rest of the paper is structured as follows: in Sect. 2, we give an overview of $\mathcal{P}revent$. In Sects. 4 and 5, we provide the details of, respectively, constructing the monitor, and the run-time monitoring algorithm. We define a measure to assess the prediction accuracy and validate the performance of the monitor using hypothesis testing in Sect. 6. Finally, we provide the empirical evaluation of $\mathcal{P}revent$ on two case studies in Sect. 7.

2 An Overview of $\mathcal{P}revent$

The key idea in $\mathcal{P}revent$ is to finitely extend the execution trace using a prediction model, and check the extended path against the specification property.

[1] Available at https://bitbucket.org/rbabaeecar/prevent/.

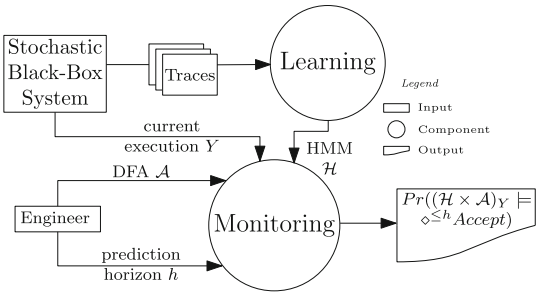


Fig. 1. The overview of $\mathcal{P}revent$ framework.

The prediction model is obtained from *iid* sample traces collected from the past executions of the system. The prediction model enables the monitor to estimate the extensions that satisfy or violate the given property within a finite horizon, that is represented as the maximum length of the finite extensions. This gives the monitor the ability to detect a property violation before its occurrence.

An overview of $\mathcal{P}revent$ is shown in Fig. 1. The two main components of $\mathcal{P}revent$, *learning* and *monitoring* are described below:

Learning. We use the sample traces to train HMM using Baum-Welch algorithm [26]. The training samples represent an independent and identical distribution (*iid*) over all the execution traces of the system. The trained HMM represents the joint distribution of the paths over Σ^* and S^*, where Σ is the observation space and S is the state space of the system.

Monitoring. The monitor in our framework is the result of a bounded reachability analysis on the product of the HMM and the DFA that specifies the acceptable or unacceptable extensions by the property. The monitor is implemented as a lookup table. Each entry is a composite state that specifies a DFA state, a hidden state in the HMM, and an observation, and the probability that from the current state the system will satisfy or violate the property in a bounded number of steps. The current hidden states maintain a history of the previous observations (the prefix Y in Fig. 1). The monitor updates its estimation of the current state by running the Viterbi approximation to obtain $(\mathcal{H} \times \mathcal{A})_Y$. The output of the monitor is therefore $Pr(\mathcal{H} \times \mathcal{A}_Y \models \diamond^{\leq h} Accept)$, where h is the finite horizon, or the maximum length of the extensions that are estimated by the monitor. Since $\mathcal{H} \times \mathcal{A}$ has a small size, the probability results of the reachability analysis can be computed off-line for all the states $(\mathcal{H} \times \mathcal{A})$, and for $1 \leq h \leq H_{MAX}$, and stored in a table. The value of H_{MAX} represents the maximum length of the extensions that the monitor needs to predict the evaluation of the property, and can be obtained empirically from the execution samples.

3 Preliminaries

In this section, we briefly introduce the used definitions and notations.

A probability distribution over a finite set S is a function $P : S \to [0,1]$ such that $\sum_{s \in S} P(s) = 1$. We use $X_{1:\tau}$ to denote a sequence $x_1, x_2, \ldots, x_\tau$ of values of a random variable X, and use u and w to resp. denote a finite and infinite path.

Hidden Markov Model (HMM): HMM is the joint distribution over $X_{1:\tau}$, the sequence of one state variable, and $Y_{1:\tau}$, the sequence of observations (both with identical lengths). The joint distribution is such that $Pr(y_i \mid X_{1:i}, Y_{1:i}) = Pr(y_i \mid x_i)$ for $i \in [1..\tau]$ (the current observation is conditioned only on the current state), and $Pr(x_i \mid X_{1:i-1}, Y_{1:i-1}) = Pr(x_i \mid x_{i-1})$ for $i \in [1..\tau]$ (the current state is only conditioned on the previous hidden states). We use π to denote the initial probability distribution over the state space, i.e., $Pr(x_1) = \pi(x_1)$. As a result, an HMM can be defined with three probability distributions:

Definition 1 ***(HMM).*** *A finite discrete Hidden Markov Model (HMM) is a tuple $\mathcal{H} : (S, \Sigma, \pi, T, O)$, where S is the non-empty finite set of states, Σ is the non-empty finite set of observations, $\pi : S \to [0,1]$ is the initial probability distribution over S, $T : S \times S \to [0,1]$ is the transition probability, and $O : S \times \Sigma \to [0,1]$ is the observation probability. We use $\Theta_{\mathcal{H}}$ to denote π, T, and O.*

Discrete-Time Markov Chains (DTMC). We use Discrete-Time Markov Chain (DTMC) for reachability analysis necessary to construct our monitor.

Definition 2 ***(DTMC).*** *A Discrete-Time Markov Chain (DTMC) is a tuple $\mathcal{M} : (S, \Sigma, \pi, \mathbf{P}, L)$, where S is a non-empty finite set of states, Σ is a non-empty finite alphabet, $\pi : S \to [0,1]$ is the initial probability distribution over S, $\mathbf{P} : S \times S \to [0,1]$ is the transition probability, such that for any $s \in S$, $P(s, \cdot)$ is a probability distribution, and $L : S \to \Sigma$ is the labeling function.*

Deterministic Finite Automaton: We use Deterministic Finite Automaton (DFA) to describe the extensions of a prefix.

Definition 3 ***(DFA).*** *A Deterministic Finite Automaton (DFA) is a tuple $\mathcal{A} : (Q, \Sigma, \delta, q_I, F)$, where Q is a set of finite states, Σ is a finite alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function determining the next state for a given state and symbol in the alphabet, $q_I \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states ($\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ denotes the language of a DFA $\mathcal{A}$).*

4 Monitor Construction

A monitor is a finite-state machine (FSM) that consumes the output of the system execution sequentially, and produces the evaluation of a given property at each step, typically as a Boolean value [4]. The monitor in our framework is still an FSM, in the form of a look-up table, that instead of Boolean values produces a value in [0,1]. The value indicates the probability of the extensions that satisfy

or violate the specification, assuming that the property is currently not satisfied/violated. These probability values are the result of a bounded reachability analysis on the product of the trained HMM and the DFA.

In the rest of this section, we describe how an HMM is built using standard *Expectation-Maximization* (EM) learning technique [6] (Sect. 4.1), describe the product model of HMM and DFA as a DTMC used to perform the reachability analysis (Sect. 4.2), and, our monitor construction approach (Sect. 4.3).

4.1 Training HMM

We use *Maximum Likelihood Estimation* (MLE) technique [29] to train an HMM. The log-likelihood function $L(\Theta)$ of the HMM $\mathcal{H} : (S, \Sigma, \pi, T, O)$ over an observation sequence $Y_{1:\tau}$ is defined as $L(\Theta) = \log(\sum_{X_{1:\tau}} Pr(X_{1:\tau}, Y_{1:\tau} \mid \theta))$.

Since the probability distribution over the state sequence $X_{1:\tau}$ is unknown, $L(\Theta)$ does not have a closed form [37], leaving the training techniques to heuristics such as EM. One well-known EM technique for training an HMM is *Baum-Welch* algorithm [26] (BWA), where the training alternates between estimating the distribution over the hidden state variable, $Q : X \rightarrow [0,1]$, with some fixed choice for Θ (*Expectation*), and maximizing the log-likelihood to estimate the values of Θ by fixing Q (*Maximization*) [28].

The *Expectation* phase in BWA computes $Pr(X_t = s \mid Y, \Theta)$ and $Pr(X_t = s, X_{t+1} = s' \mid Y, \Theta)$ for $s, s' \in S$ through *forward-backward* algorithm [26]. *Maximization* is performed on a lower bound of $L(\Theta)$ using Jensen's inequality: $L(\Theta) \geq Q(X) \log Pr(X_{1:\tau}, Y_{1:\tau} \mid \Theta) - Q(X) \log Q(X)$.

Since the second term is independent of Θ [28], only the first term is maximized in each iteration: $\Theta^{(k)} = \text{argmax}_\Theta Q(X) \log Pr(X_{1:\tau}, Y_{1:\tau} \mid \Theta^{(k-1)})$.

The training starts with random initial values for $\Theta^{(0)}$, and consequently running the forward-backward algorithm to update the parameters of the model:

$$\pi^*(s) = Pr(X_1 = s \mid Y, \Theta) \qquad T^*(s,s') = \frac{\sum_{t=1}^{\tau} Pr(X_t = s, X_{t+1} = s' \mid Y, \Theta)}{\sum_{t=1}^{T} Pr(X_t = s \mid Y, \Theta)}$$

$$O^*(s,o) = \frac{\sum_{t=1}^{\tau} \#(Y_t = o) \cdot Pr(X_t = s \mid Y, \Theta)}{\sum_{t=1}^{T} Pr(X_t = s \mid Y, \Theta)}$$

BWA is essentially a gradient-decent approach, thus its outcome is highly sensitive to the initial values of Θ [37].

We use the Bayesian Information Criterion (BIC) [7] to choose the number of hidden states. BIC assigns a score to a model according to its likelihood but also penalizes models with more parameters to avoid over-fitting: $BIC(\mathcal{H}) = log(n)|\Theta| - 2L(\Theta)$, where $|\Theta| = |S|^2 + |S||\Sigma|$, and n is the size of training sample.

4.2 The Product of the Prediction Model and the Specification

From each state of the trained HMM, the monitor needs to expand the observed execution, u, and predict expected value of the given property. The expansion of

u is based on a DFA that specifies good and bad extensions of u. The monitor maintains the configurations of both the DFA and the trained HMM by creating the product of the two models [39,41]:

Definition 4 ***(The Product of an HMM and a DFA).*** *Let $\mathcal{H} = (S, \Sigma, \pi, T, O)$ and $\mathcal{A} = (Q, \Sigma, \delta, q_I, F)$ respectively be an HMM and a DFA. We define the DTMC $\mathcal{M}_{\mathcal{H}\times\mathcal{A}} : (S' = S \times Q \times \Sigma, \{Accept\}, \pi', \mathbf{P}, L)$ as follows:*

$$\pi'(s,q,o) = \begin{cases} \pi(s) & \text{if } q \in q_I \\ 0 & \text{otherwise} \end{cases} \qquad L(s,q,o) = \begin{cases} \{Accept\} & \text{if } q \in F \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathbf{P}((s,q,o),(s',q',o')) = \begin{cases} T(s,s') \cdot O(s',o') & \text{if } \delta(q',o) = q \\ 0 & \text{otherwise} \end{cases}$$

4.3 Constructing Monitor with Bounded Prediction Horizon

The monitor's purpose is to estimate the probability of all the finite extensions of length at most h that satisfy a given property. The variable h is a positive integer we call the *prediction horizon*. Let $\sigma_0\sigma_1 \cdots \sigma_t$ be the extension of a finite path that ends at the state σ ($\sigma_0 = \sigma \in S'$), such that $L(\sigma_t) = Accept$ in the product model $\mathcal{M}$ ($\sigma_i = (s_i, q_i, o_i)$ is the composite state of the product model $\mathcal{M}$, for all $0 \leq i \leq t$). The monitor's output is $Pr(\sigma_0\sigma_1 \cdots \sigma_t)$, $t \leq h$, which is computed by performing the following reachability analysis on $\mathcal{M}$ [1]: $Pr(\sigma \models \Diamond^{\leq h} Accept)$.

In order to compute this probability we adopt the transformation from [16]:

$$\mathbf{P}_{Acc}(\sigma,\sigma') = \begin{cases} 0 & \text{if } L(\sigma) = Accept \text{ and } \sigma \neq \sigma' \\ 1 & \text{if } L(\sigma) = Accept \text{ and } \sigma = \sigma' \\ \mathbf{P}(\sigma,\sigma') & \text{otherwise} \end{cases} \tag{1}$$

The transformation (1) allows us to recursively compute $Pr(\sigma \models \Diamond^{\leq h} Accept)$ as follows: $Pr(\sigma \models \Diamond^{\leq h} Accept) = \sum_{\sigma'} \mathbf{P}_{Acc}(\sigma,\sigma') Pr(\sigma' \models \Diamond^{\leq h-1} Accept)$ (2).

This is essentially the *transient probability* for $\sigma_0 \cdots \sigma_h w$ [16], that is, starting from σ_0 the probability of being at state σ_h (i.e., after h steps), such that $L(\sigma_h) = Accept$ ($w \in \Sigma^\omega$ is any infinite extension of the path). The probability measure of $\sigma_0 \cdots \sigma_h w$ is based on the prefix $\sigma_0\sigma_1 \cdots \sigma_h$ and can be written as the joint probability distribution of the hidden state variable and that of the observation.

Computing (2) for all the states at runtime is not practical, due to multiplications of large and typically sparse matrices [16]. Instead, for all $t \leq h$ we compute the probabilities off-line and store them in a table $MT(\sigma,t)$, where $MT(\sigma,t) = Pr(\sigma \models \Diamond^{\leq t} Accept)$. Our monitor, thus, is transformed into a look-up table with the size at most $O(|S| \times |Q| \times |\Sigma| \times h)$.

5 Run-Time Monitoring with Viterbi Approximation

For each state $\sigma = (s,q,o)$ the monitor needs to estimate the hidden state s (q is derivable from o). We employ the Viterbi algorithm to find the most likely hidden state during monitoring.

```
1  MONITOR(Y, H, A, h)
   inputs : Execution observation Y, HMM H = (S, Σ, π, T, O), DFA A = (Q, Σ,
            δ, q_I, F), prediction horizon h
   output: Pr((H × A)_Y ⊨ ◇^{≤h} Accept)
2  begin
3      Construct the monitor table MT(H,A, Σ, h)
4      foreach s ∈ S do v(s) ← O(s,Y_1)π(s)// Initialize the Viterbi vector
5      i ← 1, t ← h, q ← q_I                  // t is the horizon index
6      forall Y_i ∈ Y  do
7          s ← argmax_s v(s)
8          q ← δ(q,Y_i)
9          output MT((s,q,Y_i), t)            // Output the prediction
10         if q ∈ F or t = 0 then t ← h else t ← t − 1
11         forall s ∈ S  do // Updating the next Viterbi vector
12             v_next(s) ← O(s,Y_{t+1}) max_{s'}(v(s')T(s',s))
13         v ← v_next, i ← i + 1
```

Runtime monitoring procedure using Viterbi approximation.

For an observation sequence $Y = Y_{1:\tau}$, Viterbi algorithm [10,38] derives $X^*_{1:\tau} = \text{argmax}_{X_{1:\tau}} Pr(X_{1:\tau}|Y,\Theta)$, so-called the *Viterbi path*. Let $v_t(s)$ be the probability of the Viterbi path ending with state s at time t: $v_t(s) = O(s,Y_t) \max_{s' \in S}(v_{t-1}(s')T(s',s))$ (3).

To find X^*_t at step t, the monitor only requires $v_{t-1}(s')$ for all $s' \in S$. Therefore, we can obtain X^*_t by using only two vectors that maintain the values of $v_t(s)$ and $v_{t-1}(s)$ (we call them *Viterbi vectors*).

Procedure MONITOR demonstrates our runtime monitoring algorithm. We assume that the monitor table MT is already constructed as described in Sect. 4 (line **3**). Line **4** initialize the Viterbi vector. The *horizon index* t stores the prediction horizon at each iteration (initialized to h at the beginning – line **5**). Each iteration of the *for* loop in lines **6**–**13** is over one observation in the sequence Y. For each observation Y_i, the configuration (s,q,Y_i) (lines **7**–**8**) combined with t gives us the index to retrieve the probability value in the monitor table (line **9**). If the path is not accepted by the DFA, the monitor shrinks its horizon index by one (t is decremented — line **10**). Each time that the observed path is accepted by the DFA, the horizon index is reset to h (line **10**), for the prediction of the next extension. Similarly, once the prediction horizon has reached zero, i.e., the property is not satisfied within the given prediction horizon, the horizon index is reinitialized to h. At the end, the Viterbi vector is updated for the next iteration in lines **11**–**13** according to (3).

In each monitoring iteration (the loop in lines **6-13**), reading the value from the monitor table MT is constant time. For a trained model with k hidden states, updating the Viterbi vector requires $O(k)$ operations of finding maximums, which can be improved to $lg(k)$ using a Max-Heap. Therefore, each monitoring iteration is of $O(k\, lgk)$ in execution time. The space complexity is mainly bounded by the size of the monitor table and the Viterbi vectors: $O(k\, h)$.

6 Prediction Evaluation

In this section, we first define a lower bound on the prediction error of the monitor on a single trace, and then use two-sided hypothesis testing to evaluate the average prediction performance on a set of testing samples. Finally, we exploit the hypothesis testing results to find an empirical lower bound of the horizon.

6.1 Prediction Error

Let $(o_i \cdots o_{i+\lambda_i(\mathcal{A})})$ be an extension of length $\lambda_i(\mathcal{A})$ at point i that is accepted by a given DFA $\mathcal{A}$, i.e., $(o_i \cdots o_{i+\lambda_i}) \in \mathcal{L}(\mathcal{A})$ (for brevity, we use λ_i instead of $\lambda_i(\mathcal{A})$). Recall that the monitor's output at point i is the probability of all the extensions of length at most h that are accepted by $\mathcal{A}$ ($Pr(\sigma_i \models \Diamond^{\leq h} Accept)$). For any $\lambda_i \leq h$ we have: $Pr(\sigma_i \models \Diamond^{\leq h} Accept) \geq Pr(\sigma_i \cdots \sigma_{i+\lambda_i} \models Accept) \implies \lambda_i \times Pr(\sigma_i \models \Diamond^{\leq h} Accept) \geq \lambda_i \times Pr(\sigma \cdots \sigma_{i+\lambda_i} \models Accept)$.

We define $\hat{\lambda}_i = \lambda_i \times Pr(\sigma_i \models \Diamond^{\leq h} Accept)$ as the expected value of λ_i estimated by the monitor. Therefore, we can obtain the following minimum error of the prediction at point i: $\varepsilon_i^{min} = \lambda_i - \hat{\lambda}_i$.

Notice that since $\lambda_i \geq \hat{\lambda}_i$, ε^{min} is always positive. If there is no k, $i < k < \lambda_i$ such that $(o_i \cdots o_{i+k}) \in \mathcal{L}(\mathcal{A})$, i.e., $(o_i \cdots o_{i+\lambda_i})$ is the minimal extension that is accepted by $\mathcal{A}$, then $\varepsilon_{i+t}^{min} = (\lambda_i - t) - \hat{\lambda}_{\lambda_i - t}, 0 \leq t < \lambda_i \leq h$, where t is the horizon index in Monitor. As a result, the value of ε_i^{min} can be computed on-the-fly.

In our implementation, we assume that there exists at least one point $k \leq h$ such that $(o_i \cdots o_{i+k}) \in \mathcal{L}(\mathcal{A})$; otherwise, ε_i^{min} is not well-defined, and the prediction accuracy can not be verified. If such a point does not exist, we can extend the prediction horizon by increasing h such that there is at least one accepting extension in the trace. The rest of the path after the last point in which the trace is accepted by $\mathcal{A}$ is discarded as there is no observation to compare the prediction and compute the error.

In the following, we give an empirical evaluation of the monitor's prediction using hypothesis testing which leads to an empirical lower bound for h.

6.2 Empirical Evaluation Using Hypothesis Testing

To assess the performance of the prediction, aside from the execution trace, we use hypothesis testing on a set of test samples.

Let $\Lambda = \frac{1}{\tau} \sum_{i=1}^{\tau} \lambda_i$ be the random variable that represents the mean of all λ_i values, for $1 \leq i \leq \tau$. Notice that for *iid* samples, the value of Λ for a trace is independent of that value for the other traces.

Let $\bar{\lambda}_M$ be the estimation of Λ by the monitor over a set of monitored traces, and $\bar{\lambda}$ be the mean of Λ on a separate set of n *iid* samples with variance ν. We test the accuracy of the prediction using the following two-sided hypothesis test $H_0 : \bar{\lambda}_M = \bar{\lambda}$. Using confidence α, we use student's t-distribution to test H_0: $\frac{\bar{\lambda} - \bar{\lambda}_M}{\frac{\sqrt{\nu}}{n}} \leq t_{n-1,\alpha}$. Given the mean of the length of the shortest finite extensions in the test sample we can use it to obtain a lower bound for h: $h \geq \bar{\lambda} - t_{n-1,\alpha} \frac{\sqrt{\nu}}{n}$.

That is, *the prediction horizon* h must be at least as large as the mean of the length of the extensions in the test sample that are accepted by $\mathcal{A}$.

7 Case Studies

We evaluate *Prevent* on two case studies: (1) randomized dining philosophers from PRISM [27], which includes the original algorithm, and a modified version that we introduce specifically for evaluating *Prevent*, (2) QNX Neutrino kernel traces collected from the flight control software of a hexacopter. We show the estimation of *good* and *bad* extensions in the randomized dining philosophers and hexacopter traces, respectively, each of which represents one of the most commonly used property patterns in Dwyer *et al.* [11]'s survey: *response* pattern in the randomized dining philosophers algorithm, and the *absence* pattern for monitoring a regular safety property [1] in the flight control of a hexacopter. The implementation of monitoring in both experiments is conducted off-line.

7.1 Randomized Dining Philosopher

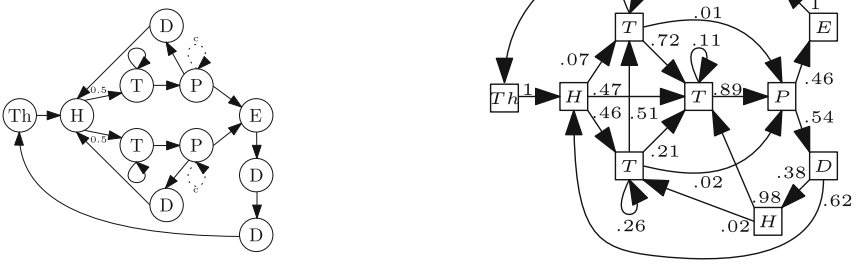
We adapt Rabin and Lehmann [25]'s solution to the dining philosophers problem that has the characteristics of a stochastic system to be trained using HMM. We also present a modification of their algorithm, which represents a generic form of decentralized on-line resource allocation [36], where our monitoring solution can be seen as a component of the *liveness enforcement supervisory* [19].

We consider the classic form of the problem, where philosophers are in a ring topology, and are selected for execution by a fair scheduler. Figure 2a demonstrates a state diagram of one philosopher, with Th, H, T, P, D, and E representing the philosopher to be, respectively, *thinking*, *hungry*, *trying*, *picking* a fork, *dropping* a fork, and *eating*. A philosopher starts at (Th), and immediately transitions to (H)[2]. Based on the outcome of a fair coin, the philosopher then chooses to pick the left or the right fork if they are available, and moves to (T). If the fork is not available the philosopher remains at (T) until it is granted access to the fork. The philosopher moves to (E), if the other fork is available; otherwise, the philosopher drops the obtained fork, moves to (D), and eventually transitions back to (H). After the philosopher finishes eating, it drops the forks in an arbitrary order (D), and moves back to (Th). The algorithm is deadlock-free but lockouts are still possible [25].

Our modification of the algorithm is to add a self-transition at (P): a philosopher does not drop the first obtained fork with probability c, i.e., it stays at (P), which is shown with dotted lines in Fig. 2a (the transition from (P) to (D) has the probability $1-c$, which is not shown in the figure). This modification enables the philosopher to control its *waiting time*, the period between when it becomes hungry for the first time after thinking, and when it eats. A higher value of c

[2] For simplicity, we remove a self-transition to (Th); however, unlike [9] we do not merge the states (Th) and (H) because we want to distinguish between the incoming transitions to (Th) and (H) in computing the waiting time.

means that, instead of going back to (H), the philosopher is more likely to stay at (P) so that as soon as the other fork is available it will eat. It is not difficult to observe that as long as there is at least one philosopher with $c \neq 1$, the symmetry that causes the deadlock [25] will eventually break, and the algorithm remains deadlock-free. In a distributed real-time system, where each philosopher represents a process with unfixed deadlines, changing the value of c enables the processes to dynamically adjust their waiting time according to their deadlines.



(a) The states of one philosopher. The dotted self-transitions display our modification.

(b) The trained HMM of one philosopher, in a system with three philosophers.

Fig. 2. Training an HMM for the monitored philosopher in a program with 3 philosophers.

The purpose of our experiments is to implement a monitor that observes the outputs of a single philosopher, and predicts a potential starvation (lockout) by estimating the extensions that leads to *eating*.

Predicting Starvation at Run-Time. We use Matlab HMM toolbox to train HMMs, and 100 *iid* samples collected from the implementation of our modified version, with $c = 1$ for all philosophers except the one that is being monitored[3]. The trained model presents the behavioral signature of the system when a *longer waiting time* is likely. The size of HMM (i.e., the number of hidden states) is chosen based on the *BIC score* of each model with different sizes (see Sect. 4.1). Figure 2b demonstrates the trained HMM of one philosopher that is constructed from the traces of a 5-s execution of three philosophers. The trained model reflects the distribution of the prefixes in the training sample, which in turn is determined by how the scheduler as well as other philosophers behaved during training (i.e., resolving non-determinism of the model). For instance, multiple consecutive *try*s in the training sample create several states in the trained HMM,

[3] We tweaked the implementation in https://ti.tuwien.ac.at/tacas2015/ [14].

Table 1. Prediction results on 100 test samples.

N	Size of HMM	BIC ($+e03$)	h^{min}	Size of MT	$\bar{\lambda}_M$	$mean(\varepsilon^{min})$
3	17	25.1	9.94	360	9.30	1.75
4	14	11.9	5.49	180	5.30	1.28
5	10	10.1	6.36	154	6.16	0.80
6	14	7.69	5.61	180	5.17	1.05
7	16	6.09	4.28	170	3.84	1.06
8	10	5.42	4.94	110	4.32	1.33
9	14	4.83	3.15	120	2.77	0.92
10	10	4.40	4.31	110	3.84	0.97

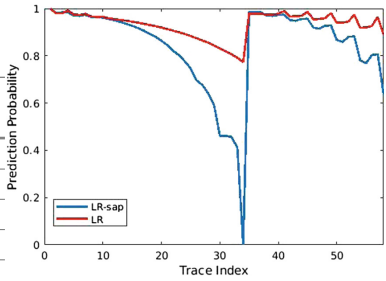


Fig. 3. The comparison of the prediction results from two trained models.

each emitting the symbol (T), but only one has a high probability to transition to (P) and the others model the state where the philosopher can not pick a fork. Finite extensions that we consider in the prediction are based on the following regular expression: $(\neg hungry)^*(hungry(\neg eat)^* eat(\neg hungry)^*)^*$.

Figure 3 gives a comparison between the prediction results ($h = 33$) of two trained models, one trained using the samples from the original implementation (LR) and the other one trained from the samples of our modified version (LR-sap), both containing three philosophers. The monitored trace is synthesized in a way that it does not contain any *eat*, and up to point 33 the philosopher is only at state (T). After that the philosopher frequently picks and drops a fork. When the last event of a prefix is *pick*, compared to when it ends with any other observations, the philosopher has a higher chance to reach *eat* (e.g., with probability 0.98 at point 35); however, since HMM maintains the history of the trace, a prefix with frequent (*pick*, *drop*) one after another shows a decline in the probability of observing *eat* (e.g., with probability of 0.8 at point 57). The results in Fig. 3 demonstrate that the model trained on the *bad* extensions (LR-sap) provides an under-approximation of the model that is trained on the *good* traces (LR), thus, producing more false positives.

The summary of our results is displayed in Table 1. We use PRISM to perform the reachability analysis on the product of the trained HMM and the DFA. The size of the product model is equal to the size of the HMM, as each state in the trained HMMs emits exactly one observation. The *minimum prediction horizon* (h^{min}) is obtained empirically from 100 test samples. We choose the prediction horizon to be three times as large as h^{min} during monitoring. The average of the estimated length of the acceptable extensions by the monitor is shown as $\bar{\lambda}_M$, and the mean of the error on the entire testing set is denoted by $mean(\varepsilon^{min})$. On average, the monitor predicts the next *eat* (within the prediction horizon) with one step error. The monitor is not able to detect the waiting periods that approximately are longer than $3 \times h^{min} \pm 1$. Increasing the prediction horizon decreases the error, with the cost of a larger monitor table (MT). The value of $\bar{\lambda}_M$ is influenced by the total number of discrete events produced by

the monitored philosopher. With more philosophers $\bar{\lambda}_M$ decreases because the monitored philosopher, and hence, the monitor, are scheduled less often.

7.2 Hexacopter Flight Control

In this section, we apply *Prevent* to detect injected faults from QNX Neutrino's [24] kernel calls. The traces are obtained using QNX `tracelogger` during the flight of a hexacopter[4]. The vehicle is equipped with an autopilot, but can be controlled manually using a remote transmitter. The autopilot system uses a cascaded PID controller. QNX's microkernel follows message-passing architecture, where almost all the processes (even the kernel processes) communicate via sending and receiving messages that are handled by the kernel calls `MSG-SENDV`, `MSG-RECEIVEV`, and `MSG-REPLY`. Figure 4a shows a sub-trace of the kernel call sample from the hexacopter flight control system.

In this case study, we inject faults by introducing an interference process, with the same priority as the autopilot process, that simply runs a while-loop to consume CPU time. The interference process abrupts message-passing between the processes of the same or lower priorities, causing a kernel call to handle the error (typically due to a timeout) and to unblock the sender (shown as event `MSG_ERROR` in Fig. 4a). The purpose of the monitor is to predict the existence of an interference process by only observing the kernel calls.

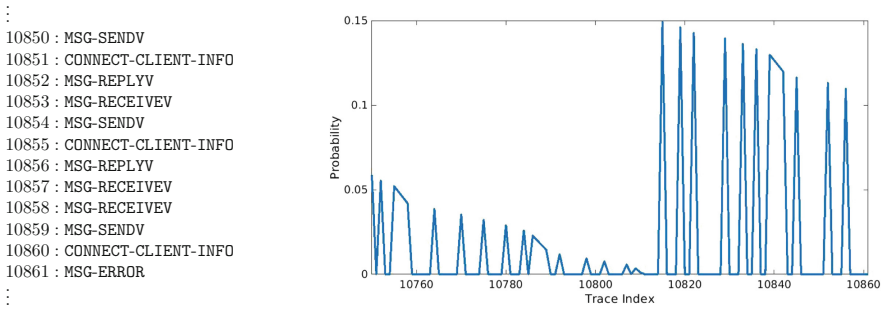
We use SFIHMM [8] on an Intel Xeon 2.40 GHz 128 GB RAM machine with Debian 9.3 to train an HMM from 1-s of the auto-pilot execution, with the intervening process in full effect. The HMM with the minimum BIC has 19 states. The regular expression $(\neg\texttt{MSG\_ERROR})^*\ (\texttt{MSG\_ERROR})\Sigma^*$ is used to generate the finite extensions that contain the bad prefixes of the property $\Box\neg\texttt{MSG\_ERROR}$.

The monitor's prediction on part of the trace generated from another scenario, where the interference process is partially in effect and started executing in the middle of the flight, is depicted in Fig. 4. The event `MSG_ERROR` is emitted at index 10,861, and the probability of the prefix that contains `MSG_ERROR` within next 50 steps is 0.15 at index 10,815. That means that the monitor predicts the message error with %15 chance, almost 45 steps before its occurrence. The points where the probability is *zero* is because the monitor was not able to correctly estimate the hidden state of the model. More training samples are required to enable the monitor to estimate the correct state of the model. In our case, three consecutive instances of `MSG_RECEIVEV` did not appear in the training sample, hence, the prefix can not be associated to any state of the model by the monitor.

8 Related Work

There have been several proposals to define semantics of LTL properties on the finite paths [20]; however, to the best of our knowledge, our approach is the first

[4] Full system description is available at https://wiki.uwaterloo.ca/display/ESGDAT/QNX+Hexacopter+Flight+Control+Dataset.



(a) A sub-trace of the kernel calls, 20 steps before the event `MSG_ERROR`.

(b) The monitor prediction 50 steps before the event `MSG_ERROR`.

Fig. 4. The monitoring of $\Box\neg$`MSG_ERROR` on the flight control trace with the interference process.

one in verifying finite paths based on the extensions obtained from a trained HMM. HMMs have been recently used in run-time monitoring of CPSs [2,13,30, 31,33,35,40]. Sistla *et al.* [31] propose an *internal* monitoring approach (i.e., the property is specified over the hidden states) using specification automata and HMMs with infinite states. Learning an infinite-state HMM is a harder problem than the finite HMMs, but does not require inferring the size of the model [5].

The notion of *acceptance accuracy* and *rejection accuracy* in [30] are the complement to our notion of prediction error. According to their definition, our Viterbi approximation generates a threshold *conservative* monitor for any regular safety property and regular finite horizon. The analytical method in [33] to find an upper bound for the timeliness of a monitor can be applied to *Prevent* to find an upper bound for the prediction horizon.

Several works focus on efficiently estimating the internal states of an HMM at runtime using particle filtering [13,35]. Particle filtering uses weights based on the number of particles in each state, and updates the weights in each observation. Viterbi algorithm provides the most likely state, as an over-approximation. Adaptive Runtime Verification [2] couples state estimation [35] with feed-back control loop to generate several monitors that run on different frequencies. These works are orthogonal to our framework and can be combined with *Prevent*.

Learning models for verification is executed on Markov Chain models [18,22]. HMMs are trained in [14] for statistical model checking. Our work focuses on predictive monitors using a similar technique. We also provide assessments for evaluating the learned model and inferring its size.

9 Conclusion

We introduced *Prevent*, a predictive run-time monitoring framework for properties with finite regular extensions. The core part of *Prevent* involves learning

a model from the traces, and constructing a tabular monitor using reachability analysis. The monitor produces a quantitative output that represents the probability that from the current state, the system satisfies a property within a finite horizon. The current state is estimated using Viterbi algorithm. We defined an empirical evaluation of the prediction using the expected length of the extension of the execution that satisfies the property. In future, we are interested in exploring other evaluation methods, including comparing the prediction results of the trained model with those of the *complete* model by applying *abstraction* [15].

We provided preliminary evaluation of our approach on two case studies: the randomised dining philosophers problem, and the flight control of a hexacopter. In both cases, the trained models are extracted from *bad* traces, thus, the monitor has a tendency to produce false positives. An interesting modification to our approach, which would reduce the number of false positives, is to involve a mixture of trained models based on both good and bad traces, and only employing ones that have a higher likelihood to generate the current execution trace.

Lastly, an implementation of $\mathcal{P}revent$ with the application of on-line learning methods (such as state merging or splitting techniques [21,34]) is necessary to apply the framework to the real-world scenarios.

References

1. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
2. Bartocci, E., Grosu, R., Karmarkar, A., Smolka, S.A., Stoller, S.D., Zadok, E., Seyster, J.: Adaptive runtime verification. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 168–182. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35632-2_18
3. Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly, but how ugly is ugly? In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 126–138. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77395-5_11
4. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput. **20**(3), 651–674 (2010)
5. Beal, M.J., Ghahramani, Z., Rasmussen, C.E.: The infinite hidden Markov model. In: Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, pp. 577–584. MIT Press, Cambridge (2001)
6. Bilmes, J.A.: A gentle tutorial of the EM algorithm and its applications to parameter estimation for Gaussian mixture and hidden Markov models. Technical report TR-97-021, International Computer Science Institute, Berkeley, CA (1997)
7. Claeskens, G., Hjort, N.L.: Model Selection and Model Averaging. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, Cambridge (2008)
8. DeDeo, S.: Conflict and computation on Wikipedia: a finite-state machine analysis of editor interactions. Futur. Internet **8**(3), 31 (2016)
9. Duflot, M., Fribourg, L., Picaronny, C.: Randomized dining philosophers without fairness assumption. Distrib. Comput. **17**(1), 65–76 (2004)

10. Durbin, R., Eddy, S.R., Krogh, A., Mitchison, G.J.: Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press, Cambridge (1998)
11. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 1999 International Conference on Software Engineering, pp. 411–420 (1999)
12. Falcone, Y., Fernandez, J.-C., Mounier, L.: Runtime verification of safety-progress properties. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 40–59. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04694-0_4
13. Kalajdzic, K., Bartocci, E., Smolka, S.A., Stoller, S.D., Grosu, R.: Runtime verification with particle filtering. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 149–166. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40787-1_9
14. Kalajdzic, K., Jegourel, C., Lukina, A., Bartocci, E., Legay, A., Smolka, S.A., Grosu, R.: Feedback control for statistical model checking of cyber-physical systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 46–61. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_4
15. Katoen, J.-P.: Abstraction of probabilistic systems. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 1–3. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75454-1_1
16. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72522-0_6
17. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. **78**(5), 293–303 (2009)
18. Mao, H., Chen, Y., Jaeger, M., Nielsen, T.D., Larsen, K.G., Nielsen, B.: Learning probabilistic automata for model checking. In: 2011 Eighth International Conference on Quantitative Evaluation of Systems, pp. 111–120, September 2011
19. Moody, J., Antsaklis, P.: Supervisory Control of Discrete Event Systems Using Petri Nets. The International Series on Discrete Event Dynamic Systems. Springer, New York (1998). https://doi.org/10.1007/978-1-4615-5711-1
20. Morgenstern, A., Gesell, M., Schneider, K.: An asymptotically correct finite path semantics for LTL. In: Bjørner, N., Voronkov, A. (eds.) LPAR 2012. LNCS, vol. 7180, pp. 304–319. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28717-6_24
21. Mukherjee, K., Ray, A.: State splitting and merging in probabilistic finite state automata for signal representation and analysis. Sig. Process. **104**, 105–119 (2014)
22. Nouri, A., Raman, B., Bozga, M., Legay, A., Bensalem, S.: Faster statistical model checking by means of abstraction and learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 340–355. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_28
23. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57 (1977)
24. Qnx neutrino rtos. http://blackberry.qnx.com/en/products/neutrino-rtos/neutrino-rtos. Accessed 14 Aug 2017
25. Rabin, M.O., Lehmann, D.: The advantages of free choice: a symmetric and fully distributed solution for the dining philosophers problem. In: Roscoe, A.W. (ed.) A Classical Mind, pp. 333–352. Prentice Hall International (UK) Ltd., Hertfordshire (1994)
26. Rabiner, L.R.: A tutorial on hidden Markov models and selected applications in speech recognition. Proc. IEEE **77**(2), 257–286 (1989)

27. Radomised dining philosophers case study. http://www.prismmodelchecker.org/casestudies/phil.php. Accessed 24 Jan 2018
28. Roweis, S.T., Ghahramani, Z.: A unifying review of linear Gaussian models. Neural Comput. **11**(2), 305–345 (1999)
29. Shalev-Shwartz, S., Ben-David, S.: Understanding Machine Learning: From Theory to Algorithms. Cambridge University Press, Cambridge (2014)
30. Sistla, A.P., Srinivas, A.R.: Monitoring temporal properties of stochastic systems. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 294–308. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78163-9_25
31. Sistla, A.P., Žefran, M., Feng, Y.: Monitorability of stochastic dynamical systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 720–736. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_58
32. Sistla, A.P., Žefran, M., Feng, Y.: Runtime monitoring of stochastic cyber-physical systems with hybrid state. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 276–293. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_21
33. Sistla, A.P., Zefran, M., Feng, Y., Ben, Y.: Timely monitoring of partially observable stochastic systems. In: HSCC, 17th International Conference (Part of CPS Week), pp. 61–70 (2014)
34. Stolcke, A., Omohundro, S.M.: Best-first model merging for hidden Markov model induction. CoRR, abs/cmp-lg/9405017 (1994)
35. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime verification with state estimation. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 193–207. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_15
36. Tanenbaum, A.S., van Steen, M.: Distributed Systems - Principles and Paradigms, 2nd edn. Pearson Education, London (2007)
37. Terwijn, S.A.: On the learnability of hidden Markov models. In: Adriaans, P., Fernau, H., van Zaanen, M. (eds.) ICGI 2002. LNCS (LNAI), vol. 2484, pp. 261–268. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45790-9_21
38. Viterbi, A.J.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Trans. Inf. Theory **13**(2), 260–269 (1967)
39. Wilcox, C.M., Williams, B.C.: Runtime verification of stochastic, faulty systems. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 452–459. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_34
40. Yavolovsky, A., Žefran, M., Sistla, A.P.: Decision-theoretic monitoring of cyber-physical systems. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 404–419. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_25
41. Zhang, L., Hermanns, H., Jansen, D.N.: Logic and model checking for hidden Markov models. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 98–112. Springer, Heidelberg (2005). https://doi.org/10.1007/11562436_9
42. Zheng, X., Julien, C., Podorozhny, R., Cassez, F., Rakotoarivelo, T.: Efficient and scalable runtime monitoring for cyber-physical system. IEEE Syst. J. **PP**(99), 1–12 (2017)

Applications

Formal Verification of Platoon Control Strategies

Adnan Rashid[1(✉)], Umair Siddique[2], and Osman Hasan[1]

[1] School of Electrical Engineering and Computer Science (SEECS), National University of Sciences and Technology (NUST), Islamabad, Pakistan
{adnan.rashid,osman.hasan}@seecs.nust.edu.pk
[2] Department of Computing and Software, McMaster University, Hamilton, Canada
siddiu3@mcmaster.ca

Abstract. Recent developments in autonomous driving, vehicle-to-vehicle communication and smart traffic controllers have provided a hope to realize platoon formation of vehicles. The main benefits of vehicle platooning include improved safety, enhanced highway utility, efficient fuel consumption and reduced highway accidents. One of the central components of reliable and efficient platoon formation is the underlying control strategies, e.g., constant spacing, variable spacing and dynamic headway. In this paper, we provide a generic formalization of platoon control strategies in higher-order logic. In particular, we formally verify the stability constraints of various strategies using the libraries of multivariate calculus and Laplace transform within the sound core of HOL Light proof assistant. We also illustrate the use of verified stability theorems to develop runtime monitors for each controller, which can be used to automatically detect the violation of stability constraints in a runtime execution or a logged trace of the platoon controller. Our proposed formalization has two main advantages: (1) it provides a framework to combine both static (theorem proving) and dynamic (runtime) verification approaches for platoon controllers; and (2) it is inline with the industrial standards, which explicitly recommend the use of formal methods for functional-safety, e.g., automotive ISO 26262.

Keywords: Autonomous driving · Platoon control
Formal verification

1 Introduction

Autonomous cars seem to be just around the corner, as most of the car manufacturers (e.g., Tesla, BMW, Toyata, Nissan, Ford, Jaguar Land Rover, etc.) and even silicon valley players (e.g., Intel and Nvidia) claim that fully autonomous vehicles will be on the road around 2020 [1,2]. Such a speedy development in autonomous driving is motivated by the fact that the autonomous cars will be more safe and crashless than the human driven cars. For example, the human error is to blame for up to 90% of the 1.2 million deaths that occur each year from

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 223–238, 2018.
https://doi.org/10.1007/978-3-319-92970-5_14

car accidents around the world [3]. Like various fields of science and engineering, the developments in autonomous driving have opened the doors to many other interesting fields, for example, *automated vehicle platooning* is one of the most benefiting fields.

A *platoon* is a group of vehicles (as shown in Fig. 1) that travels in a close proximity to one another, nose-to-tail, at highway speeds. Vehicle platoons have

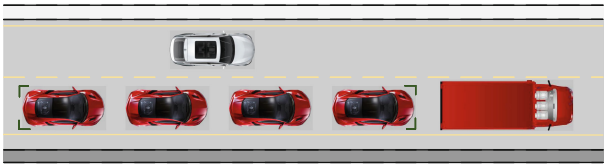


Fig. 1. Platoon of vehicles

been proposed since at least the early 1980's even before we had wireless communication, global positioning system (GPS) and commercially available radar sensors. However, given the exceptional capabilities and reliability of the autonomous cars, vehicle platooning can become a reality using a mix of available technologies such as drive-by-wire steering [4], radar cruise control [5] and lane keep assist systems [6] to name a few. Some of the main advantages of the vehicle platooning are increased road capacity, reduction in drag and improved fuel economy, improved traffic congestion strategies [7] and reduced roadside accidents due to the autonomous features, e.g., collision detection [8] and automatic emergency braking [9].

The stability of the automated vehicles in a platoon, individually or as a group, depends on the interaction of the vehicles and is vital for an uninterrupted flow of traffic and a better throughput. A *stable platoon* ensures that the vehicles should not collide with each other while maintaining a safe inter-vehicle spacing bound. In practice, the stability of platoon is ensured by two types of controllers, i.e., autonomous and non-autonomous [10]. The autonomous controllers use the on-board sensors for determining the speed and position of the connected vehicles, whereas the non-autonomous controllers are based on other forms of the inter-vehicle communication. Furthermore, communication amongst controllers is either unidirectional or bidirectional, based on the information shared between the neighbouring vehicles. Similarly, various strategies can be used for the platoon control, such as constant spacing, variable spacing and variable time headway.

Traditionally, the platoon controllers are analyzed using informal approaches including paper-and-pencil based proofs [10] and numerical simulations [11]. These informal approaches have known limitations when used in safety-critical domains, for example, missing assumptions and even wrong derivations in hand-driven proofs, and inherent incompleteness of the numerical algorithms, respectively. Considering these facts, it is natural to think about complementing traditional analysis approaches with formal methods for developing reliable platoon

controllers. In this direction, model checking has been used to verify the high-level models of the platoon controllers using the temporal logic based properties [12–14]. In all these approaches, the authors considered the vehicles platoon and their controllers as a discrete-time system by modeling them as automata and verified their properties, such as safety and inter-vehicle spacing bound properties. Thus, these model checking based analysis lacks the physical analysis of the platoon, which requires modeling and reasoning of control strategies using differential equations and their frequency domain stability analysis using Laplace transform. Similarly, Mashkoor et al. [15] used higher-order logic to formally reason about the cyber-physical transportation system. The authors used random variables to model the unpredictable elements of the system and formally conducted a probabilistic analysis of the transportation system without considering the dynamic aspects of the system. In this paper, we propose a higher-order logic based framework to formally model and verify the stability of various types of platoon controllers using the HOL Light proof assistant [16]. Consequently, we utilize the verified results to construct monitors, which ensure the platoon stability at runtime. The main reasons for using higher-order logic and HOL Light include the expressibility to represent the platoon controllers, which are modeled using differential equations in time-domain and the Laplace transform in frequency-domain. Moreover, HOL Light has the smallest trusted core (i.e., approximately 400 lines of Ocaml code) amongst all other HOL proof assistants and the underlying logic kernel has been verified in the CakeML project [17].

The main contributions of the paper are as follows:

- Deep embedding based formalization of platoon controller types, configurations and strategies along with the associated differential equations based functional models.
- Formal derivation of the Laplace domain transfer functions using the formalized libraries of multivariate calculus [18] and the Laplace transform [19,20] in the HOL Light proof assistant.
- Formal verification of the platoon control strategies based on the formal models of various controllers.
- Development of the stability monitors for each type of the controllers and demonstration of their violation detection capability on a pseudo-randomly generated traces of a platoon controller.

The source code of our HOL Light development is available for download at [21] and thus can be used by the other researchers and engineers interested in the design and verification of the platoon controllers.

The rest of the paper is organized as follows: Sect. 2 presents an overview of the HOL Light proof assistant along with the formalization of the Laplace transform. Section 3 provides the formal modeling of the platoon controller and its stability. We provide the formal verification of the platoon control strategies and the stability constraints in Sect. 4. Section 5 describes the construction of the stability monitors. Finally, Sect. 6 concludes the paper and highlights some future research directions.

2 Preliminaries

This section presents a brief introduction to the HOL Light proof assistant and its multivariate calculus and the Laplace transform theories, which are extensively used in the rest of the paper.

2.1 Theorem Proving and HOL Light Proof Assistant

Theorem proving is a widely adapted formal verification technique, which is concerned with constructing the proofs of the mathematical theorems using a computer program (called *theorem prover or proof assistant*) [22]. Theorem proving systems have been commonly used for verifying the properties of the software and hardware systems. For example, a hardware designer can certify a digital circuit by modeling its behavior by some predicates and verifying its different properties using Boolean algebra. Similarly, a mathematician can verify the transitive property of the ordering of real numbers using some basic axioms of real numbers theory. These properties are expressed as theorems using some logic, such as propositional, first-order or higher-order logic, based on the required expressiveness. For example, using the higher-order logic is advantageous over the first-order logic as it provides the additional quantifiers and is more expressive as well. Moreover, higher-order logic can better describe the complex mathematical concepts including multivariate calculus, transcendental functions and topological spaces. Once such a mathematical theory is developed inside a proof assistant, we say that it is formalized.

HOL Light [16] is an interactive theorem proving environment for constructing the mathematical proofs. The main implementation of HOL Light is done in a functional programming language, Objective CAML (OCaml), which is originally developed to automate the mathematical proofs [23]. The logical kernel of HOL Light is of approximately 400 lines of OCaml code and its main components are its types, terms, theorems, rules of inference, and axioms. A theory in HOL Light consists of types, constants, definitions, axioms and theorems. The HOL Light theories are ordered in a hierarchical fashion and the child theories can inherit the types, definitions and theorems of the parent theories. Every new theorem has to be verified based on the primitive inference rules and basic axioms or already verified theorems present in HOL Light, which ensures the soundness of this technique.

2.2 Multivariable Calculus and Laplace Transform Theories

HOL Light provides an extensive support for the analysis of physical systems based on multivariate calculus theories, which include derivatives, integration, transcendental theory, topology, vector analysis and Laplace transform theory. Table 1 presents some definitions from the Laplace transform theory of HOL Light, which includes the Laplace transform, Laplace existence and the exponential-order conditions, and the differential equation of order n. Interested readers can refer to [19,20] for more details about this theory. It is extensively used in our proposed verification of the platoon control strategies for the automated vehicles.

Table 1. Laplace transform

Mathematical Form	Formalized Form
Laplace Transform	
$\mathcal{L}[f(t)] = F(s) = \int_0^\infty f(t)e^{-st}dt,\ s \in \mathbb{C}$	⊢ ∀ s f. **laplace_transform** f s = integral {t | &0 <= drop t} (λt. cexp (−−(s ∗ Cx (drop t))) ∗ f t)
Laplace Existence	
f is piecewise smooth and is of exponential order on the positive real line	⊢ ∀ s f. **laplace_exists** f s ⇔ (∀ b. f piecewise_differentiable_on interval [lift (&0),lift b]) ∧ (∃ M a. Re s > drop a ∧ exp_order_cond f M a)
Exponential-order Condition	
There exist a constant a and a positive constant M such that $\|f(t)\| \leq Me^{at}$	⊢ ∀ f M a. **exp_order_cond** f M a ⇔ &0 < M ∧ (∀ t. &0 <= t ⇒ norm (f (lift t)) <= M ∗ exp (drop a ∗ t))
Differential Equation of Order n	
Differential Equation = $\sum_{k=0}^{n} \alpha_k \frac{d^k f}{dt^k}$	⊢ ∀ n f t. **diff_eq_n_order** n lst f t = vsum (0..n) (λk. EL k [$\alpha_1, \alpha_2, ..., \alpha_k$] ∗ higher_order_derivative k f t)

3 Formal Modeling of Platoon Controller and Stability

In this section, we present the formal modeling of a platoon controller based on its types, configurations and the underlying strategies along with the concept of the platoon stability.

3.1 Formalization of Platoon Controller

The connected vehicles in a platoon are widely characterized by the controllers, which are mainly responsible for their automated operation. The platoon controllers are generally of two types namely autonomous and non-autonomous.

- *Autonomous controllers* use the on-board sensors for determining the speed and position of the connected vehicles.
- *Non-autonomous controllers* are based on some other form of the inter-vehicle communication.

The information sharing among the neighbouring vehicles is either unidirectional or bidirectional depending upon the configuration of the platoon controllers.

- *Unidirectional configuration* allows a controller to use the information about the relative distance and velocity of only the preceding vehicle.
- *Bidirectional controller* can access the information about the relative distance and velocity of both the proceeding and preceding vehicles by considering their individual masses.

The autonomous controllers can adapt different strategies to maintain the stable operation of the platoon along the highway. In general, controllers utilize three strategies namely constant spacing, variable spacing and variable time-headway.

- *Constant spacing policy* requires that each vehicle maintains a constant distance (spacing) with its preceding vehicle in a platoon.
- *Variable spacing policy* allows a variable inter-vehicle spacing, which depends on the velocity of the vehicles in a platoon. For example, a faster moving vehicle creates more inter-vehicle space between itself and its proceeding vehicle. It is also known as the *constant time headway spacing*.
- *Variable time headway* policy imposes constraints on the relative velocity rather than the absolute velocity of the vehicle in contrast to the constant time headway spacing policy, which results into large inter-vehicle spaces and thus decreases the throughput of the highway traffic.

In our formalization, we model the types of the controller, its configurations and strategies as enumerated datatype using the built-in define_type mechanism in HOL Light.

```
type controller_type = autonomous | non_autonomous
type configuration = unidirectional | bidirectional
type strategy = constant_spacing | variable_spacing | var_time_headway
```

We model a platoon as a tuple $(x, n, m, k, c, ch, vd, h0, ca, cd)$, where the description and the type of each parameter is given in Table 2. Indeed, these parameters characterize various physical aspects of the vehicles in a platoon (e.g., the horizontal distance x, the number of vehicles in a platoon n and the mass of a vehicle m). In HOL Light, we formalize the platoon tuple $(x, n, m, k, c, ch, vd, h0, ca, cd)$ and controller tuple $(controller\_type, configuration, strategy, platoon)$ as type abbreviations:

```
type_abbrev platoon:(x × n × m × k × c × h × ch × vd × h0 × ca × cd)
type_abbrev controller:(controller_type × configuration × strategy × platoon)
```

It is important to note that `platoon` contains a unique mass m, which implies that we only consider a platoon with identical vehicles as shown in Fig. 1.

In order to ensure that the given parameters of a platoon indeed represent a valid platoon, we formalize the associated constraints as a predicate is_valid_platoon (Definition 1). For example, the mass m should always be greater than 0 and the number of vehicles in a platoon should be greater than 1.

Definition 1. Valid Platoon

```
⊢ is_valid_platoon (x,n,m,k,c,h,ch,vd,h0,ca,cd) ⇔ 0 < m ∧ 0 < k ∧ 0 < c ∧
                0 < h ∧ 0 < ch ∧ 0 < vd ∧ 0 < h0 ∧ 0 < ca ∧ 0 < cd ∧ 1 < n
```

Table 2. Data types for platoon parameters

Parameter description	Type
Horizontal distance	$\mathsf{x}{:}\mathbb{N} \to (\mathbb{R} \to \mathbb{C})$
Number of vehicles	$\mathsf{n}{:}\mathbb{N}$
Mass of a vehicle	$\mathsf{m}{:}\mathbb{R}$
Disturbance constant	$\mathsf{k}{:}\mathbb{R}$
Fluctuations constant	$\mathsf{c}{:}\mathbb{R}$
Time headway	$\mathsf{h}{:}\mathbb{R}$
Fluctuations due to time headway	$\mathsf{ch}{:}\mathbb{R}$
Desired platoon speed	$\mathsf{vd}{:}\mathbb{R}$
Nominal value of time headway	$\mathsf{h0}{:}\mathbb{R}$
Additional fluctuations with respect to platoon leader	$\mathsf{ca}{:}\mathbb{R}$
Additional fluctuations with respect to "virtual" mass	$\mathsf{cd}{:}\mathbb{R}$

3.2 Formalization of the Platoon Stability

The *stability* is an important property of a vehicle platoon, which describes the capability of a platoon to attenuate the oscillations introduced by the leader or any other vehicle in the platoon. In general, such oscillations can be considered in terms of various signals. e.g., the position error between the vehicles or the relative acceleration of connected vehicles. In this paper, we consider the notion of stability with respect to the position error between the vehicles. Formally, a platoon is stable if any oscillation with respect to the position error diminishes out as it propagates towards the tail of the platoon. The platoon stability in longitudinal direction is mathematically expressed as a norm condition on spacing errors in the frequency domain, as given in the following equation:

$$\left|\left|\frac{z_n(i\omega)}{z_{n-1}(i\omega)}\right|\right| < 1, \quad n = 2, 3, 4, ... \tag{1}$$

where z_{n-1} is the spacing error between the vehicle $n-1$ and its proceeding vehicle n, i.e., it is the deviation from the desired inter-vehicle spacing for vehicle $n-1$. If x_{n-1} is the inter-vehicle spacing between the vehicle $n-1$ and its preceding vehicle $n-2$ and x_n is the inter-vehicle spacing between the vehicle n and its preceding vehicle $n-1$, then the spacing error between vehicles $n-1$ and n is given by $z_{n-1} = x_{n-1} - x_n$. Similarly, z_n represents the spacing error between the vehicle n and its proceeding vehicle $n+1$, i.e., $z_n = x_n - x_{n+1}$. In case of all the desired inter-vehicle spacings are same, i.e., $x_n = x_{n-1} = ... = x_1$, then this leads to the zero spacing errors, i.e., $z_n = z_{n-1} = ... = z_1 = 0$. We formalize platoon stability in HOL Light as follows:

Definition 2. Stable Platoon

$$\vdash \forall\, \mathsf{s}\ \mathsf{x}\ \mathsf{y}.\ \textbf{transfer\_function}\ \mathsf{s}\ \mathsf{x}\ \mathsf{y} \Leftrightarrow \frac{\mathsf{laplace\_transform\ y\ s}}{\mathsf{laplace\_transform\ x\ s}}$$

$$\vdash \forall\, \omega\ \mathsf{x}\ \mathsf{y}.\ \textbf{frequency\_response}\ \omega\ \mathsf{x}\ \mathsf{y} \Leftrightarrow \mathsf{transfer\_function}\ (\mathsf{i}\omega)\ \mathsf{x}\ \mathsf{y}$$

$$\vdash \forall\, \omega\ \mathsf{z}.\ \textbf{is\_stable\_platoon}\ \omega\ \mathsf{z}\ \mathsf{n} \Leftrightarrow$$

$$\left\| \mathsf{frequency\_response}\ \omega\ (\lambda \mathsf{t}.\ \mathsf{z}\ (\mathsf{n}))\ (\lambda \mathsf{t}.\ \mathsf{z}\ (\mathsf{n}\ -\ 1)) \right\| < 1$$

where the predicate is_stable_platoon accepts the parameters $\mathsf{z}{:}\mathbb{N} \rightarrow (\mathbb{R} \rightarrow \mathbb{C})$, which represents the complex-domain representation of the spacing error, angular frequency $\omega{:}\mathbb{R}$ and number of vehicles n, and returns the condition that the complex norm of the transfer function at $s = i\omega$, i.e., $\frac{Z_n(i\omega)}{Z_{n-1}(i\omega)}$ is always less than 1 for every vehicle in the platoon.

This concludes our fundamental formalization of the platoon controller and the corresponding stability. We build upon the concepts, formalized in this section, to formalize various control strategies and verify their correctness in the next section.

4 Formal Verification of the Platoon Control Strategies

In this section, we first present the formalization of an autonomous unidirectional controller with constant spacing policy. Indeed, the main intention is to demonstrate the formalization steps, i.e., formal modeling of the controller dynamics in higher-order logic, formalization of the necessary constraints, and the formal verification of the stability theorem. Building upon these steps, we next present its generalization to all types of controllers along with the verification of a generalized stability theorem.

4.1 Autonomous Unidirectional Controller

Generally, the dynamics of platoon controllers are characterized by a set of differential equations, which interrelate the parameters of the platoon. The schematic representation of the platoon of vehicles having autonomous unidirectional controller with constant spacing policy is depicted in Fig. 2. It consists of n interconnected vehicles of identical masses, i.e., $m_1 = m_2 = ... = m_{n-1} = m_n = m$. The parameters k and c are the disturbance and fluctuation constants, representing the control gains on the relative position and velocity, respectively. Similarly, the parameter u represents the force required by the first vehicle to move forward in the platoon. The mathematical representation of this platoon controller's dynamics are given as the following equation set:

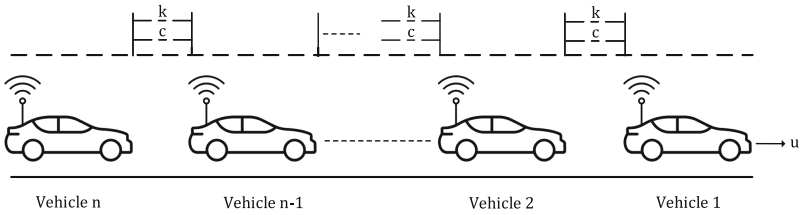


Fig. 2. Autonomous unidirectional controller with constant spacing

$$
\begin{aligned}
\frac{dx_1}{dt} &= v_1, \quad \frac{dv_1}{dt} = \frac{u}{m}, \quad \frac{dx_2}{dt} = v_2 \\
\frac{dv_2}{dt} &= \frac{k}{m}x_1 - \frac{k}{m}x_2 + \frac{c}{m}v_1 - \frac{c}{m}v_2 \\
&\vdots \\
\frac{dx_{n-1}}{dt} &= v_{n-1} \\
\frac{dv_{n-1}}{dt} &= \frac{k}{m}x_{n-2} - \frac{k}{m}x_{n-1} + \frac{c}{m}v_{n-2} - \frac{c}{m}v_{n-1} \\
\frac{dx_n}{dt} &= v_n \\
\frac{dv_n}{dt} &= \frac{k}{m}x_{n-1} - \frac{k}{m}x_n + \frac{c}{m}v_{n-1} - \frac{c}{m}v_n
\end{aligned}
\tag{2}
$$

where the variables x and v represent the inter-vehicle spacing and velocity of platoon vehicles, respectively. Overall, the set of differential equations characterize the dynamics of n vehicles in the platoon depicted in Fig. 2. We can rewrite Eq. (2) in a compact form by eliminating the variable v and representing it in the form of spacing error, i.e., z as:

$$
\frac{d^2 z_n}{dt^2} + \frac{c}{m}\frac{dz_n}{dt} + \frac{k}{m}z_n = \frac{c}{m}\frac{dz_{n-1}}{dt} + \frac{k}{m}z_{n-1}, \quad n = 2, 3, 4, \ldots \tag{3}
$$

We formally model this controller in HOL Light as follows:

Definition 3. Unidirectional Controller with Constant Spacing

```
⊢ ∀ k c m ch n vd ca cd h h0 x.
  control_uni_cs (autonomous,unidirectional,constant_spacing,
                                  ((x,n,m,k,c,h,ch,vd,h0,ca,cd):platoon)) t ⇔
    let z_{n-1} = (λt. x (n - 1) t - x (n) t) and
        z_n = (λt. x (n) t - x (n + 1) t) in
                  D^2 [k/m; c/m; 1] z_n = D^1 [k/m; c/m] z_{n-1}
```

where the operators $\mathcal{D}^1$ and $\mathcal{D}^2$ represent the first-order and second-order complex-valued derivatives in HOL Light, respectively, and thus can be obtained by instantiating $n = 1$ and $n = 2$ in the predicate diff_eq_n_order, given in Table 1.

We next model some physical constraints associated with the controller model control_uni_cs, which include differentiability, existence of the Laplace transform and zero-initial conditions for parameters z_{n-1} and z_n, as given in Definition 4.

Definition 4. Constraints for a Platoon having Autonomous Unidirectional Controller

```
⊢ ∀ x n s c m k.
  constraints_uni_cs x n s c m k ⇔
      let z_{n-1} = (λt. x (n - 1) t - x (n) t) and
          z_n = (λt. x (n) t - x (n + 1) t) in
            (∀t. differentiable_higher_derivative [2,1] [z_{n-1},z_n]) ∧
                 laplace_exists_higher_deriv [2,1] [z_{n-1},z_n] s ∧
                 zero_initial_conditions [1,0] [z_{n-1},z_n] ∧
                 non_zero_tf_uni_cs z_{n-1} s c m k
```

where the first two conjuncts provide the differentiability and the Laplace existence conditions for the second-order and first-order derivatives of the spacing errors z_{n-1} and z_n, respectively. Similarly, the next conjunct imposes the zero-initial conditions for the spacing errors z_{n-1} and z_n, respectively. Finally, the last conjunct ensures that the transfer function does not include the singularities, i.e., the points at which the denominator of the transfer function becomes infinite or undefined. Mathematically, it is described as $s^2 + \frac{c}{m}s + \frac{k}{m} \neq 0$.

Our next step is to formally verify that the platoon controller model control_uni_cs implies the platoon stability for any number of vehicles. The main purpose of this verification is twofold: (1) identify the stability constraints in terms of the platoon parameters, and (2) utilize verified stability constraints to ensure the stability of a given platoon at any time instant. Indeed this step requires the instantiation of platoon parameters with concrete values (i.e., number of vehicles $n = 10$, mass $m = 1000\,\text{kg}$, etc.). We verify the following universally quantified stability theorem in HOL Light.

Theorem 1. Stability of a Platoon having Autonomous Unidirectional Controller

```
⊢ ∀ k c m ch n vd ca cd h h0 x w.
  let s = iω and
      p = ((x,n,m,k,c,h,ch,vd,h0,ca,cd):platoon) and
      z = (λ n t. x (n) t - x (n + 1) t) in
  0 < ω ∧ 2k/m < ω² ∧ valid_platoon p ∧ constraints_uni_cs x n s c k m ∧
  ∀ t. control_uni_cs (autonomous,unidirectional,constant_spacing,p) t
       ⟹ is_stable_platoon ω z n
```

The main proof of Theorem 1 consists of the following steps: (1) rewriting with the Definitions 1–4, (2) complex arithmetic reasoning and (3) the verification of Lemma 1, which transforms the time-domain model of the platoon

controller control_uni_cs into its equivalent frequency-domain representation, i.e., transfer function. The verification of Lemma 1 is quite involved due to the reasoning about the Laplace transform in HOL Light [19]. The formal statement of Lemma 1 is given as follows:

Lemma 1. Model Implies Transfer Function

```
⊢ ∀ k c m ch n vd ca cd h h0 x s.
  let p = ((x,n,m,k,c,h,ch,vd,h0,ca,cd):platoon) and
      z_{n-1} = (λt. x (n - 1) t - x (n) t) and
      z_n = (λt. x (n) t - x (n + 1) t) in
      valid_platoon p ∧ constraints_uni_cs x n s c k m ∧
      ∀ t. control_uni_cs (autonomous,unidirectional,constant_spacing,p) t
        ⟹ transfer_function s z_n z_{n-1} = (c/m s + k/m) / (s^2 + c/m s + k/m)
```

4.2 Generalized Platoon Controller

We formally model various types of platoon control strategies as given in Table 3. We also formalize the physical constraints and verify the stability for each control strategy along the same lines as that of autonomous unidirectional controller presented in Sect. 4.1. Finally, we package them in an inductive predicate gen_platoon_control, which takes two parameters, i.e., controller and time t and returns the predicate describing the physical behavior of the controller. For example, for controller (autonomous,unidirectional,constant_spacing,platoon), the inductive predicate gen_platoon_control returns control_uni_cs[1].

Finally, we verify a general theorem, which describes the stability constraints for any type of the controller *cc*, as follows:

Theorem 2. Stability of a Platoon

```
⊢ ∀ (cc:controller) s.
  let s = iω and
      p = (x,n,m,k,c,h,ch,vd,h0,ca,cd):platoon and
      cc = (ct,cg,sg,p) and
      z = (λn  t. x (n) t - x (n + 1) t) in
      gen_stability_physical_constraints cc s ω ∧ ∀ t. gen_platoon_control cc t
        ⟹ is_stable_platoon ω z n
```

where the predicate gen_stability_physical_constraints encapsulates the stability and physical constraints of all types of controllers in our formalization.

[1] We have omitted the formal definition of gen_platoon_control for the sake of conciseness, however, interested reader can find the formal definition and HOL Light code on the project's webpage [21].

The formal proof of Theorem 2 is based on induction on cc:controller and further induction on the controller_type, configuration and strategy along with the verified stability theorems for each control strategy (e.g., Theorem 1 for autonomous unidirectional controller presented in Sect. 4.1).

This concludes our formalization of platoon controllers in the HOL Light proof assistant. In summary, we formalized the basic notions of the platoon controllers

Table 3. Formal platoon models considering various control strategies

Autonomous Unidirectional Controller with Speed-dependent Spacing
$\vdash \forall$ k c m ch n vd ca cd h h0 x. **control_uni_vs** (autonomous,unidirectional,variable_spacing, ((x,n,m,k,c,h,ch,vd,h0,ca,cd):platoon)) t $\Leftrightarrow$ let $z_{n-1} = (\lambda t.\ x\ (n - 1)\ t - x\ (n)\ t)$ and $z_n = (\lambda t.\ x\ (n)\ t - x\ (n + 1)\ t)$ in $\mathcal{D}^2\ [\frac{k}{m}; \frac{c+k*h}{m}; 1]\ z_n = \mathcal{D}^1\ [\frac{k}{m}; \frac{c}{m}]\ z_{n-1}$
Autonomous Unidirectional Controller with Variable Time Headway
$\vdash \forall$ k c m ch n vd ca cd h h0 x. **control_uni_vth** (autonomous,unidirectional,var_time_headway, ((x,n,m,k,c,h,ch,vd,h0,ca,cd):platoon)) t $\Leftrightarrow$ let $z_{n-1} = (\lambda t.\ x\ (n - 1)\ t - x\ (n)\ t)$ and $z_n = (\lambda t.\ x\ (n)\ t - x\ (n + 1)\ t)$ in $\mathcal{D}^2\ [\frac{k}{m}; \frac{c+k*h0+k*ch*vd}{m}; 1]\ z_n = \mathcal{D}^1\ [\frac{k}{m}; \frac{c+k*ch*vd}{m}]\ z_{n-1}$
Autonomous Bidirectional Controller with Constant Spacing
$\vdash \forall$ k c m ch n vd ca cd h h0 x. **control_bi_cs** (autonomous,bidirectional,constant_spacing, ((x,n,m,k,c,h,ch,vd,h0,ca,cd):platoon)) t $\Leftrightarrow$ let $z_{n-1} = (\lambda t.\ x\ (n - 1)\ t - x\ (n)\ t)$ and $z_n = (\lambda t.\ x\ (n)\ t - x\ (n + 1)\ t)$ in $\mathcal{D}^2\ [\frac{2*k}{m}; \frac{2*c}{m}; 1]\ z_n = \mathcal{D}^1\ [\frac{k}{m}; \frac{c}{m}]\ z_{n-1}$
Autonomous Bidirectional Controller with Speed-dependent Spacing
$\vdash \forall$ k c m ch n vd ca cd h h0 x. **control_bi_vs** (autonomous,bidirectional,variable_spacing, ((x,n,m,k,c,h,ch,vd,h0,ca,cd):platoon)) t $\Leftrightarrow$ let $z_{n-1} = (\lambda t.\ x\ (n - 1)\ t - x\ (n)\ t)$ and $z_n = (\lambda t.\ x\ (n)\ t - x\ (n + 1)\ t)$ in $\mathcal{D}^2\ [\frac{2*k}{m}; \frac{2*c+k*h}{m}; 1]\ z_n = \mathcal{D}^1\ [\frac{k}{m}; \frac{c}{m}]\ z_{n-1}$
Non-autonomous Controller considering Communication of Leader's Current Velocity
$\vdash \forall$ k c m ch n vd ca cd h h0 x. **control_clcv** non_autonomous ((x,n,m,k,c,h,ch,vd,h0,ca,cd):platoon) t $\Leftrightarrow$ let $z_{n-1} = (\lambda t.\ x\ (n - 1)\ t - x\ (n)\ t)$ and $z_n = (\lambda t.\ x\ (n)\ t - x\ (n + 1)\ t)$ in $\mathcal{D}^2\ [\frac{k}{m}; \frac{c+ca}{m}; 1]\ z_n = \mathcal{D}^1\ [\frac{k}{m}; \frac{c}{m}]\ z_{n-1}$

using the new type definition and corresponding physical and stability constraints. The notable feature of our formalization is its generic nature, as we can model a platoon controller with any number of vehicles composed of basic

controller types, configurations and strategies. Moreover, the physical and stability constraints are explicitly present in our formally verified stability theorems, which may get ignored in the conventional platoon analysis and may result into an unstable platoon interrupting the traffic flow on the highways. In the next section, we describe the utilization of our verified results in HOL Light to develop stability monitors for automatically detecting the violations of the stability constraints.

5 From Verified Controller to Stability Monitors

Static formal verification approaches, such as theorem proving, provide an effective way to formally model and verify digital hardware, its underlying software, control and cyber-physical systems at an appropriate abstract level. For example, we employed higher-order logic to formalize various control strategies of a platoon due to the involvement of multivariate calculus (i.e., complex frequency domain and Laplace transform). Moreover, we formally verified some of the most important stability constraints for arbitrary platoon parameters. Indeed, this is one of the main strengths of the interactive proof assistants as compared to the simulation based analysis where verification holds only for the applied test cases and thus cannot be considered as complete. However, the verification of important properties of given system in a proof assistant does not guarantee that the system will behave as expected during the runtime operation. Indeed, the verified results in a proof assistant provide a confidence that the system will behave correctly only when the corresponding conditions are met at all times during the life-time of a system. Actually this falls under the scope of runtime verification approaches, which are light-weight formal methods to monitor the correctness of a given system with respect to a formal specification at runtime.

We demonstrate here the utilization of verified stability theorems for various control strategies to construct monitors, which are capable of detecting the violation on a given execution of the system. Consider that the behavior of a platoon controller at each time instant (called an event) is characterized by the tuple platoon and frequency w, i.e., event = ((x,n,m,k,c,h,ch,vd,h0,ca,cd),w). Thus, an execution of the platoon controller consists of the sequence of events and we model it as an event list in HOL Light. Next, we consider the autonomous unidirectional controller with constant spacing, for which the stability of the platoon is ensured if the following two conditions are met for every event in the controller execution. (1) P_1 : valid_platoon (x,n,m,k,c,h,ch,vd,h0,ca,cd) and (2) $P_2 : 0 < \omega \wedge \frac{2k}{m} < \omega^2$. In terms of temporal logic, a formal requirement to ensure the platoon stability is $\Box P_1 \wedge \Box P_2$ where $\Box$ represents *Globally* (G) or an *Always* operator in the linear temporal logic (LTL). We can model this monitor in HOL Light as (ALL P_1 execution) $\wedge$ (ALL P_2 execution) where ALL is a HOL Light function, which ensures the satisfaction of a predicate on each element of the list. Moreover, we developed a tactic MONITOR_TAC, which automatically verifies that both the predicates P_1 and P_2 holds for a given platoon controller

execution as a list of events. We tested the efficiency of the MONITOR_TAC on randomly generated executions and it can check the validity in a reasonable time. For example, on average MONITOR_TAC returns the truth (T) in 3 s on an execution of 1000 unique events.

The main purpose of the above illustration was to show that the efforts spent during the formalization within an interactive proof assistant can be complemented by the development of the monitors to ensure the correctness of the system operation at runtime, and thus closing the loop from abstract verification to the real-time monitoring on the concrete system. Our illustration only describes the off-line monitoring where the platoon controller execution is given as a logged data. However, the same monitor can be used for the online monitoring by translating the monitor as a post-condition in the actual system implementation or by generating the monitor using well-known LTL3 [24] or the rewriting-based monitoring approaches [25].

We believe that the stability monitoring can be used for the already available platoon controllers by inspecting the logged traces and by adding monitors in the early controller prototypes for quickly evaluating the correctness of the underlying algorithms. Thus, engineers working on the design and development of the platoon controllers can use the proposed framework without any prior knowledge of theorem proving and gain formally analyzed insights about the given platoon control system.

6 Conclusion and Future Work

This paper provides a framework for analyzing the platoon control strategies using both the static and dynamic verification approaches. It mainly presents the formal modeling of the platoon controller and its stability using higher-order logic. Next, the proposed formalization is used for formally verifying various platoon control strategies and their stability within the sound core of the HOL Light proof assistant. Finally, the formally verified stability theorems are used to develop the runtime monitors for each of the controllers that are used for detecting the violation of any stability constraints.

In future, we plan to formally analyze the platoon considering different connected vehicles (having different masses). We can also incorporate the stability in lateral direction and their physical constraints in our framework for the platoon stability. The other direction is to formally analyze the platoon of connected vehicles, where some of the vehicles act in a malicious manner by changing the control gain and thus the properties of the controllers. Such scenarios can compromise the safety of other vehicles on the highways and result in destabilizing the traffic flow [26]. Finally, it is interesting to consider two-dimensional platoons, which can be analyzed by combining our current framework and formalization of the z-Transform [27], which is already available in the HOL Light proof assistant.

References

1. http://www.driverless-future.com/?page_id=384 (2018)
2. https://www.technologyreview.com/s/602196/2021-may-be-the-year-of-the-fully-autonomous-car/ (2018)
3. https://phys.org/news/2017-09-self-driving-cars-road-toll.html (2018)
4. Changfu, Z., Kai, L.: Development of the drive-by-wire technology. Automobile Technol. **3**(001), 1–5 (2006)
5. Van Arem, B., Van Driel, C.J.G., Visser, R.: The impact of cooperative adaptive cruise control on traffic-flow characteristics. Trans. Intell. Transp. Syst. **7**(4), 429–436 (2006)
6. Kawazoe, H., Shimakage, M., Sadano, O., Sato, S.: Lane Keeping Assistance System and Method for Automotive Vehicle, US Patent 6,493,619, 10 December 2002
7. Fernandes, P., Nunes, U.: Platooning with IVC-enabled autonomous vehicles: strategies to mitigate communication delays, improve safety and traffic flow. Trans. Intell. Transp. Syst. **13**(1), 91–106 (2012)
8. Biswas, S., Tatchikou, R., Dion, F.: Vehicle-to-vehicle wireless communication protocols for enhancing highway traffic safety. Commun. Mag. **44**(1), 74–82 (2006)
9. Yi, J., Alvarez, L., Horowitz, R., Canudas De Wit, C.: Adaptive emergency braking control using a dynamic tire/road Friction Model. In: Decision and Control, vol. 1, pp. 456–461. IEEE (2000)
10. Eyre, J., Yanakiev, D., Kanellakopoulos, I.: A simplified framework for string stability analysis of automated vehicles. Veh. Syst. Dyn. **30**(5), 375–405 (1998)
11. Barooah, P., Mehta, P.G., Hespanha, J.P.: Mistuning-based control design to improve closed-loop stability margin of vehicular platoons. Trans. Autom. Control **54**(9), 2100–2113 (2009)
12. Kamali, M., Dennis, L.A., McAree, O., Fisher, M., Veres, S.M.: Formal verification of autonomous vehicle platooning. Sci. Comput. Program. **148**, 88–106 (2017)
13. Dolginova, E.: Safety Verification for Automated Vehicle Maneuvers. Ph.D thesis, Massachusetts Institute of Technology (1998)
14. Wongpiromsarn, T., Murray, R.M.: Formal verification of an autonomous vehicle system. In: Conference on Decision and Control (2008)
15. Mashkoor, A., Hasan, O.: Formal probabilistic analysis of cyber-physical transportation systems. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2012. LNCS, vol. 7335, pp. 419–434. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31137-6_32
16. Harrison, J.: HOL light: a tutorial introduction. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 265–269. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0031814
17. Kumar, R.: Self-compilation and Self-verification. Technical report, University of Cambridge, Computer Laboratory (2016)
18. Harrison, J.: The HOL light theory of euclidean space. J. Autom. Reasoning **50**(2), 173–190 (2013)
19. Taqdees, S.H., Hasan, O.: Formalization of laplace transform using the multivariable calculus theory of HOL-Light. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 744–758. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_50

20. Rashid, A., Hasan, O.: Formalization of transform methods using HOL Light. In: Geuvers, H., England, M., Hasan, O., Rabe, F., Teschke, O. (eds.) CICM 2017. LNCS (LNAI), vol. 10383, pp. 319–332. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62075-6_22
21. Rashid, A.: Formal Verification of Platoon Control Strategies (2018). http://save.seecs.nust.edu.pk/projects/fvpcs/
22. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press, Cambridge (2009)
23. A History of OCaml (2015). http://ocaml.org/learn/history.html
24. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. Trans. Softw. Eng. Methodol. **20**(4), 14:1–14:64 (2011)
25. Havelund, K., Rosu, G.: Monitoring programs using rewriting. In: Automated Software Engineering, pp. 135–143 (2001)
26. Dunn, D.D.: Attacker-induced Traffic Flow Instability in a Stream of Automated Vehicles. Utah State University (2015)
27. Siddique, U., Mahmoud, M.Y., Tahar, S.: On the formalization of Z-Transform in HOL. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 483–498. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_31

Exploring Properties of a Telecommunication Protocol with Message Delay Using Interactive Theorem Prover

Catherine Dubois[1], Olga Grinchtein[2(✉)], Justin Pearson[3], and Mats Carlsson[4]

[1] ENSIIE, Samovar (UMR CNRS 5157), Évry, France
catherine.dubois@ensiie.fr
[2] Ericsson AB, Stockholm, Sweden
olga.grinchtein@ericsson.com
[3] Uppsala University, Uppsala, Sweden
justin.pearson@it.uu.se
[4] RISE SICS, Stockholm, Sweden
mats.carlsson@ri.se

Abstract. An important task of testing a telecommunication protocol consists in analysing logs. The goal of log analysis is to check that the timing and the content of transmitted messages comply with specification. In order to perform such checks, protocols can be described using a constraint modelling language. In this paper we focus on a complex protocol where some messages can be delayed. Simply introducing variables for possible delays for all messages in the constraint model can drastically increase the complexity of the problem. However, some delays can be calculated, but this calculation is difficult to do by hand and to justify. We present an industrial application of the Coq proof assistant to prove a property of a 4G protocol and validate a constraint model. By using interactive theorem proving we derived constraints for message delays of the protocol and found missing constraints in the initial model.

Keywords: Testing of telecommunication protocol
Constraint programming · Formal proof · Coq

1 Introduction

We presented in [11] a constraint model of a telecommunication protocol that broadcasts public warning messages [1]. The goal was to check that the message transmission conforms to specification by analysing logs. We used the constraint modelling language MiniZinc [15] to model the protocol and to find solutions that indicated errors in logs. We used this approach to analyze both real and generated logs. Since some messages of the protocol can be delayed, introducing a delay for every message increases the complexity of the model. However, it is

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 239–253, 2018.
https://doi.org/10.1007/978-3-319-92970-5_15

possible to derive a formula for some delays, which simplifies the problem to be solved by a constraint solver. We manually derived the delays in [11], but we did not prove the correctness of the derivation.

In this work we use the Coq proof assistant [20] to derive and prove the formula for delays of some messages. By using Coq we found necessary assumptions that should be made on parameters of the constraint model. We also found missing constraints in [11]. By using Coq we are guaranteed that resulting calculations are correct. Furthermore because certain properties of the model had to be proved and derived it was more appropriate to use a proof assistant rather than a computer algebra system. The rest of the paper is structured as follows. Section 2 presents Constraint Programming and the Coq proof assistant. Section 3 is an overview of the telecommunication protocol that we analyse. Section 4 presents the constraint model on which our proofs in Coq are based. Section 5 presents a property of the protocol we explore and a new constraint model for delays. Section 6 describes proof steps in Coq.

2 Preliminaries

In this section we present very briefly, both constraint programming and the proof assistant Coq.

Constraint Programming [18] (CP) is a framework for modelling and solving combinatorial problems including verification and optimisation tasks. A constraint problem is specified as a set of *decision variables* that have to be assigned values so that the given constraints on these variables are satisfied, and optionally so that a given objective function is minimised or maximised. We use italic to distinguish *decision variables* from parameters in the constraints.

MiniZinc [15] is a constraint modelling language, which has gained popularity recently due to its high expressivity and large number of available solvers that support it. It also contains many useful modelling abstractions such as quantifiers, sets, arrays, and a rich set of global constraints. All the constraints presented in this paper are shown in a form that is very close to their MiniZinc version.

Coq [20] is an interactive proof assistant based on constructive type theory, more precisely, the calculus of inductive constructions. It also has a trustworthy kernel [2]. Coq allows the user to state theorems, write proofs that are verified according to the Curry-Howard isomorphism, and thus checking is reduced to type checking. It is also possible to write and verify algorithms. Proofs are written with the help of tactics (a.k.a. proof commands). Coq has many basic tactics, including tactics for unfolding definitions, but also more complex ones, e.g. doing arithmetic reasoning, or applying inductive proof schemes. Coq also provides a rich library of high-level tactics that automates many low level details. Coq has been used successfully in many projects of large scale such a formal proof of the four colour theorem [10] or the construction of an optimising compiler for C [14]. Interactive theorem provers differ from automatic theorem provers, such as SMT or SAT solvers, in that the user has to guide the tool to produce the proof.

3 Protocol Overview

Our case study is the Earthquake and Tsunami Warning System (ETWS) that is a part of the Public Warning System [1]. Its purpose is to broadcast emergency information to all the users in a certain area when earthquake or tsunami is imminent. We do not present all details of the protocol, but only the part that we used in the Coq development.

```
Write − ReplaceWarningRequest{
        WarningType : '0580'H
        rPer : 30
        nBR : 4
        WarningMessageContents : '41424344'H
}
```

Fig. 1. Warning message of combined type

There are three participants in the protocol that we consider: the network entity, a radio base station and the user equipments. By receiving a warning message from a network entity, the radio base station broadcasts paging messages and system information messages to the user equipments. In this work we focus on paging messages. Periodicity of paging messages depends on the type of warning message as shown in Figs. 2 and 4. Warning messages can be of three different kinds depending on its content: messages can be primary notifications, and/or secondary notifications. So the type of a warning message can be primary, secondary, or combined. A primary notification is a very simple message indicating a type of imminent danger, e.g. "earthquake", while a secondary notification message contains more detailed text data. Warning messages of combined type include both. Paging message is used to inform user equipment about the presence of primary notification and/or secondary notification.

In Fig. 1 is shown an example of the content of a warning message, where only information elements relevant to this work are included. The parameter `rPer` is in seconds and is used to calculate periodicity of paging messages of secondary notifications. The parameter `nBR` represents the number of paging messages of secondary notifications.

The periodicity of paging messages of primary notifications is equal to the default paging cycle `dPC` and the number of paging messages of primary notifications is `ndPC` that is configured in radio base station. The periodicity of paging messages of secondary notifications is a multiple of `dPC`.

Figures 2 and 4 illustrate interleaving of paging messages of primary and secondary notifications.

In Fig. 2 is shown the acquisition of paging messages by the user equipment after the radio base station receives a first warning message of primary type and then a warning message of combined type. The user equipment first reads

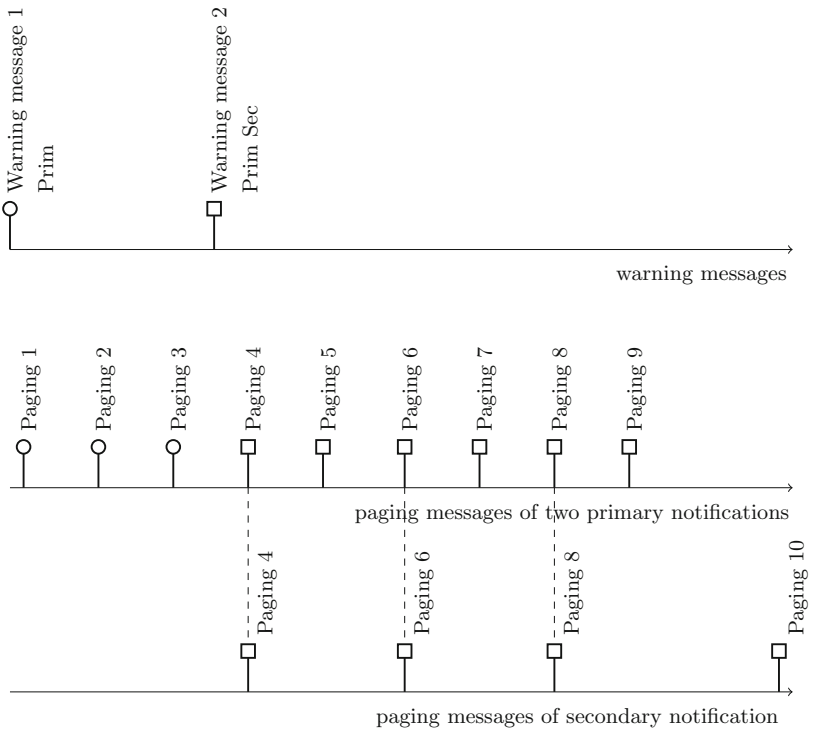


Fig. 2. An example of acquiring paging messages by user equipment transmitted by radio base station after receiving warning message of primary type and warning message of combined type. Different shapes on the top of the vertical lines represent different types of warning messages.

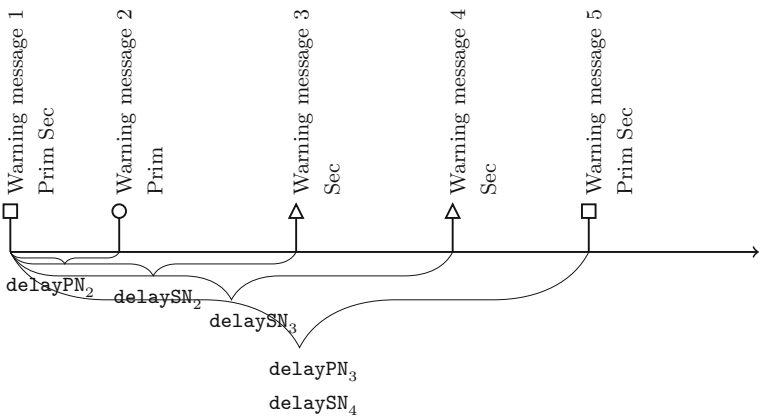


Fig. 3. Transmission of warning messages.

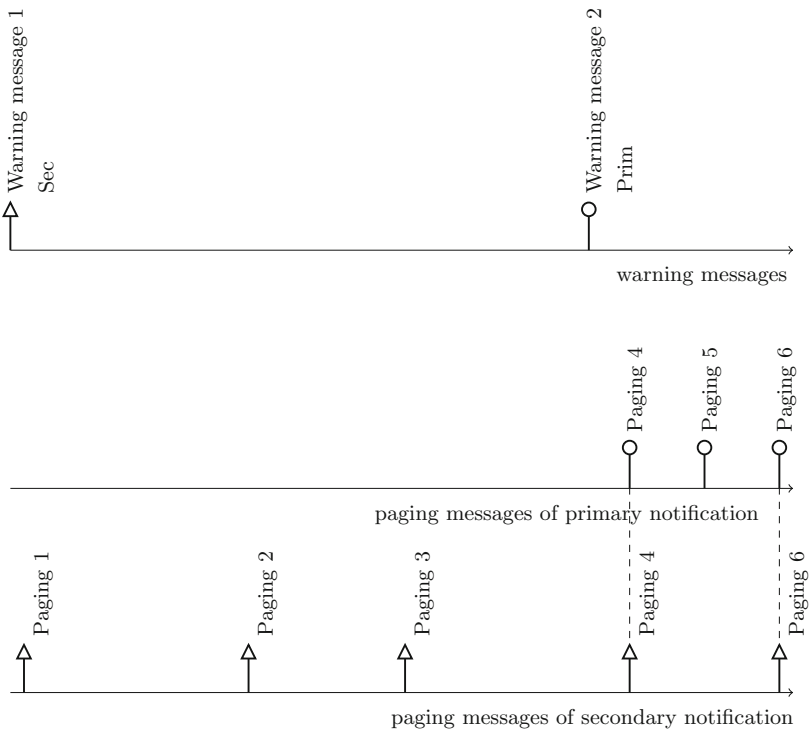


Fig. 4. An example of acquiring paging messages by user equipment transmitted by radio base station after receiving warning message of secondary type and warning message of primary type.

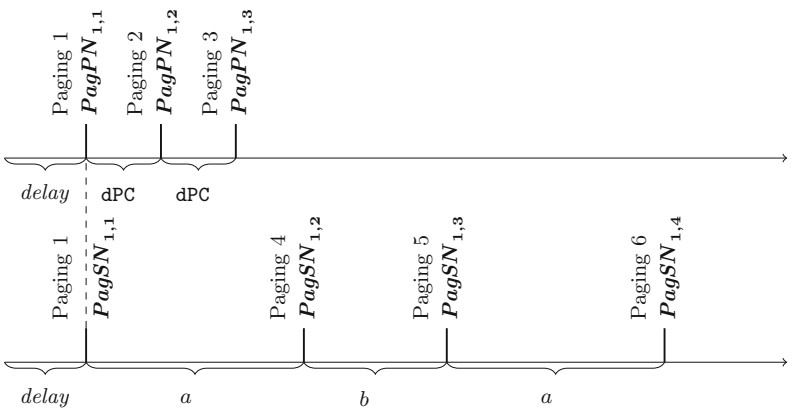


Fig. 5. An example of acquiring paging messages by user equipment transmitted by radio base station after receiving warning message of combined type.

three messages of primary notification corresponding to warning message 1. After warning message 2 is received by the radio base station, it starts to transmit paging messages of primary and secondary notifications, since warning message 2 has combined type. Figure 2 illustrates replacement of warning messages. If the radio base station receives a new warning message of primary type, while transmitting paging messages of primary notification of the previous warning message, then the radio base station starts to transmit paging messages of new primary notification. Similar replacement can occur for secondary notifications.

In Fig. 4 is shown the acquisition of paging messages by user equipment after the radio base station receives first warning message of secondary type and then warning message of primary type. The radio base station transmits paging messages of secondary type and after receiving warning message 2 it starts to transmit paging messages of primary type.

4 Constraint Model

In [11] we introduced a constraint model that had constraints on timestamps and content of messages broadcast by a radio base station, including paging messages. It is a discrete time model, since we deal with timestamps. The goal was to check that timing and content of messages in the logs comply with specification. Logs contain messages of 4 different types including paging messages. We used simulators for both network and user equipment entities, and our system under test is a radio base station. In this section we introduce a constraint model for paging messages in more details than in [11].

For each warning message we know the timestamp corresponding to the sending time. Figure 3 shows transmission of five warning messages. Since warning messages can contain primary and/or secondary notifications, we construct array `delayPN` that defines timestamps of transmission of primary notifications by network entity and array `delaySN` that defines timestamps of transmission of secondary notifications. The array `delayPN` has `nPrim` elements and the array `delaySN` has `nSec` elements. Timestamp of the first warning message is 0 and the message is of combined type. This means that $\texttt{delayPN}_1 = 0$ and $\texttt{delaySN}_1 = 0$. The second warning message is of primary type and has timestamp $\texttt{delayPN}_2$. The third warning message is of secondary type and has timestamp $\texttt{delaySN}_2$. The fourth warning message is also of secondary type and has timestamp $\texttt{delaySN}_3$. The last warning message is of combined type and hence has two equal timestamps $\texttt{delayPN}_3$ and $\texttt{delaySN}_4$.

Paging messages of primary and secondary notifications have different periodicity and we need to distinguish them in order to check that messages in the log comply with specification. Therefore we introduce a two dimensional array of correct timestamps of paging messages of primary notification *PagPN* of size $\texttt{nPrim} \cdot \texttt{ndPC}$, and another two dimensional array of correct timestamps of secondary notification *PagSN* of size $\texttt{nSec} \cdot \texttt{nBRmax}$, where `nBRmax` is the maximum number in the array `nBR` of numbers of paging messages of secondary notifications. We post constraints on these arrays which calculate periodicity of paging

messages. Periodicity of paging messages of primary notification is equal to dPC as shown in Fig. 5. The time difference between two consecutive paging messages of secondary notification depends on dPC and the repetition period rPer of the notification, and can take two different values a and b for the same notification as shown in Fig. 5. Integer decision variable *delay* in Fig. 5 represents the delay of first paging message that is the time difference between time when radio base station starts to transmit primary notification and/or secondary notification and the time user equipment reads first paging message.

The arrays delayPN and delaySN represent the timestamps of the warning messages sent to the radio base station by a network entity. Since some variable delay can occur, we introduce arrays of decision variables *delayPN50* and *delaySN50* which represent the delay of each warning message. We assume that delays are between 0 and 50 ms.

We post a constraint to guarantee that if $\texttt{delayPN}_i = \texttt{delaySN}_j$ then the equality $delayPN50_i = delaySN50_j$ holds, $1 \leq i \leq \texttt{nPrim}$ and $1 \leq j \leq \texttt{nSec}$. Since the exact number of paging messages depends on *delay*, *delayPN50* and *delaySN50*, we set *PagPN* and *PagSN* to -1 to define missing paging messages.

Constraint (1) defines the timestamp of the first paging message of the first primary notification.

$$\begin{aligned}
&\texttt{IF} \quad (\texttt{delayPN}_1 = 0)\\
&\qquad PagPN_{1,1} = 0\\
&\texttt{ELSEIF} \quad (\texttt{nPrim} > 1)\\
&\qquad ((r < \texttt{delayPN}_2 - delay \wedge PagPN_{1,1} = r) \vee\\
&\qquad (r \geq \texttt{delayPN}_2 - delay \wedge PagPN_{1,1} = -1) \vee\\
&\qquad (r \geq \texttt{delayPN}_2 - delay \wedge\\
&\qquad\quad r < \texttt{delayPN}_2 - delay + 50 \wedge PagPN_{1,1} = r \wedge\\
&\qquad\quad delayPN50_2 > r - \texttt{delayPN}_2 + delay))\\
&\texttt{ELSE}\\
&\qquad PagPN_{1,1} = r
\end{aligned} \tag{1}$$

where $r = \texttt{roundupdPC}(\texttt{delayPN}_1 - delay + delayPN50_1)$ and

$$\forall y \in \mathbb{N} \quad \texttt{roundupdPC}(y) = y + \texttt{dPC} - 1 - ((y + \texttt{dPC} - 1) \mod \texttt{dPC}) \tag{2}$$

Constraint (2) rounds y to the smallest integer that is greater or equal to y and a multiple of dPC. The constraint (2) is used to define timestamps of paging messages which are multiples of dPC.

Constraint (3) defines timestamp of first paging message of ith primary notification, $1 < i < \texttt{nPrim}$.

$$\begin{aligned}
&(\forall 1 < i < \texttt{nPrim})\\
&\qquad ((r < \texttt{delayPN}_{i+1} - delay \wedge PagPN_{i,1} = r) \vee\\
&\qquad (r \geq \texttt{delayPN}_{i+1} - delay \wedge PagPN_{i,1} = -1) \vee\\
&\qquad (r \geq \texttt{delayPN}_{i+1} - delay \wedge r < \texttt{delayPN}_{i+1} - delay + 50 \wedge PagPN_{i,1} = r \wedge\\
&\qquad\quad delayPN50_{i+1} > r - \texttt{delayPN}_{i+1} + delay))
\end{aligned} \tag{3}$$

where $r = \texttt{roundupdPC}(\texttt{delayPN}_i - delay + delayPN50_i)$.

Constraint (4) defines the timestamp of the first paging message of the last primary notification

$$\begin{array}{ll} \texttt{IF} & (\texttt{nPrim} > 1) \\ & PagPN_{\texttt{nPrim},1} = r \end{array} \tag{4}$$

where $r = \texttt{roundupdPC}(\texttt{delayPN}_{\texttt{nPrim}} - delay + delayPN50_{\texttt{nPrim}})$

By replacing `delayPN` by `delaySN`, *delayPN50* by *delaySN50* and `nPrim` by `nSec`, in (1), (3) and (4) we obtain a formula for calculating timestamps of first paging messages of secondary notification.

Constraint (5) defines the timestamp of $(k+1)$th paging message of ith primary notification, $1 \leq i < \texttt{nPrim}$

$$\begin{array}{l} (\forall 1 \leq i < \texttt{nPrim})(\forall 1 \leq k < \texttt{ndPC}) \\ \quad (PagPN_{i,k} \neq -1 \wedge PagPN_{i,k} + \texttt{dPC} < \texttt{delayPN}_{i+1} - delay + delayPN50_{i+1} \wedge \\ \qquad PagPN_{i,k+1} = PagPN_{i,k} + \texttt{dPC}) \\ \quad \vee \\ \quad (PagPN_{i,k} \neq -1 \wedge PagPN_{i,k} + \texttt{dPC} \geq \texttt{delayPN}_{i+1} - delay + delayPN50_{i+1} \wedge \\ \qquad PagPN_{i,k+1} = -1) \\ \quad \vee \\ \quad (PagPN_{i,k} = -1 \wedge PagPN_{i,k+1} = -1) \end{array} \tag{5}$$

Constraint (6) defines timestamp of $(k+1)$th paging message of the last primary notification

$$\begin{array}{l} (\forall 1 \leq k < \texttt{ndPC}) \\ \quad PagPN_{\texttt{nPrim},k+1} = PagPN_{\texttt{nPrim},k} + \texttt{dPC} \end{array} \tag{6}$$

Constraint (7) defines timestamp of $(k+1)$th paging message of jth secondary notification, $1 \leq j \leq \texttt{nSec}$, $1 \leq k < \texttt{nBR}_j$.

$$\begin{array}{l} (\forall 1 \leq j \leq \texttt{nSec})(\forall 1 \leq k < \texttt{nBR}_j) \\ \quad \texttt{IF} \quad (PagSN_{j,k} \geq 0 \wedge (j = \texttt{nSec} \vee PagSN_{j,1} + r < \\ \qquad \texttt{delaySN}_{j+1} - delay + delaySN50_{j+1})) \\ \qquad PagSN_{j,k+1} = PagSN_{j,1} + r \\ \quad \texttt{ELSE} \\ \qquad PagSN_{j,k+1} = -1 \end{array} \tag{7}$$

where $r = \texttt{roundupdPC}(\texttt{rPer}_j \cdot k - (PagSN_{j,1} - \texttt{delaySN}_j - delaySN50_j + delay))$
We based our proofs in Coq on the structure of the constraints presented in this section.

5 A New Constraint Model for Delays

In this section we focus on the property of the protocol introduced in Sect. 5.1, which helped us to design a better constraint model presented in Sect. 5.2. A new constraint model includes constraints that were missing in [11]. The constraints presented in Sect. 5.2 were obtained by interactive theorem proving with Coq.

5.1 A Property of the Protocol

Variable message delay increases complexity of the protocol drastically. A radio base station can read a warning message sent by the network entity with some delay. Introducing for each message a variable that represents delay between 0 and 50 would result in combinatorial explosion. Our goal is to find a formula to compute delay based on the structure of the constraint model of the protocol that eliminates some values and make it easier for a constraint solver to find a solution. Constraints in Sect. 4, which calculate periodicity of paging messages, contain delays *delayPN50* and *delaySN50*. If we constrain the values of *delayPN50* and *delaySN50*, the new constraint model of the protocol should have a property that the array of timestamps of paging messages *PagPN* and *PagSN* will not change.

5.2 Constraints for Delay

The major impact of the delays *delayPN50* and *delaySN50* on *PagPN* and *PagSN* is that they can increase timestamps of paging messages by $\texttt{dPC}$. We introduce constraints that define delays *delayPN50constr* and *delaySN50constr* of notification messages. For each possible value of *delayPN50* and *delaySN50* we find corresponding values *delayPN50constr* and *delaySN50constr* such that arrays of timestamps of paging messages *PagPN* and *PagSN* will not change. The following constraints are implicitly universally quantified over $1 \leq i \leq \texttt{nPrim}$, $1 \leq j \leq \texttt{nSec}$ and $1 \leq k \leq \texttt{nBR}_j$.

The timestamp of the first paging message of the primary notification should be equal to the smallest value greater or equal to $\texttt{delayPN}_i - \mathit{delay}$ and divisible by $\texttt{dPC}$. In order to increase the timestamp of the first paging message by $\texttt{dPC}$, the delay *delayPN50constr* should be the smallest value that guarantees $\texttt{roundupdPC}(\texttt{delayPN}_i - \mathit{delay} + \mathit{delayPN50constr}) > \texttt{roundupdPC}(\texttt{delayPN}_i - \mathit{delay})$. This also holds for $\texttt{delaySN}_i$ and *delaySN50constr*.

Constraint (8) defines *delayPN50constr*

$$
\begin{aligned}
&(\mathit{delayPN50}_i = 0 \wedge \mathit{delayPN50constr}_i = 0) \vee \\
&(\mathit{delayPN50}_i \geq 1 \wedge (\texttt{delayPN}_i - \mathit{delay}) \mod \texttt{dPC} = 0 \wedge \\
&\quad \mathit{delayPN50constr}_i = 1) \vee \\
&(\mathit{delayPN50}_i \geq 1 \wedge (\texttt{delayPN}_i - \mathit{delay}) \mod \texttt{dPC} > 0 \wedge \\
&\quad \texttt{roundupdPC}(\texttt{delayPN}_i - \mathit{delay}) = \\
&\quad \texttt{roundupdPC}(\texttt{delayPN}_i - \mathit{delay} + \mathit{delayPN50}_i) \wedge \\
&\quad \mathit{delayPN50constr}_i = 0) \vee \\
&(\mathit{delayPN50}_i \geq 1 \wedge (\texttt{delayPN}_i - \mathit{delay}) \mod \mathit{dPC} > 0 \wedge \\
&\quad \texttt{roundupdPC}(\texttt{delayPN}_i - \mathit{delay}) < \\
&\quad\quad \texttt{roundupdPC}(\texttt{delayPN}_i - \mathit{delay} + \mathit{delayPN50}_i) \wedge \\
&\quad \mathit{delayPN50constr}_i = \texttt{dPC} - ((\texttt{delayPN}_i - \mathit{delay}) \mod \texttt{dPC}) + 1) \qquad (8)
\end{aligned}
$$

Let $\texttt{rPerCoq}_j = \texttt{rPer}_j \cdot k - 2 \cdot \texttt{dPC}$ where k is index of paging message of jth secondary notification, $1 \leq k \leq \texttt{nBR}_j$. Let $r = \texttt{roundupdPC}(\texttt{delaySN}_j - \mathit{delay} + \mathit{delaySN50}_j)$

The timestamp of the $(k+1)$th paging message of secondary notification depends on $\texttt{rPer}_j \cdot k$ that makes constraints for the delay of secondary notification more complex than for primary notification. In order to define *delaySN50constr* we consider two cases. The first case defines the delay that increases the timestamp of the first paging message of secondary notification. The second case introduces the delay that increases the timestamp of the $(k+1)$th paging message of secondary notification.

The first case requires that the Eq. (9) holds

$$\begin{aligned}&\texttt{roundupdPC}(\texttt{rPerCoq}_j + (2 \cdot \texttt{dPC} + \texttt{delaySN}_j - \mathit{delay} - r)) = \\ &\texttt{roundupdPC}(\texttt{rPerCoq}_j + (2 \cdot \texttt{dPC} + \texttt{delaySN}_j - \mathit{delay} - r) + \mathit{delaySN50}_j) \quad (9)\end{aligned}$$

Then by replacing the variable *delayPN50* by *delaySN50* and *delayPN50constr* by *delaySN50constr* in the constraint (8), we obtain the first part of the constraint for *delaySN50constr*.

If the Eq. (9) does not hold, then we define *delaySN50constr* as

$$\begin{aligned}&(\texttt{roundupdPC}(\texttt{delaySN}_j - \mathit{delay}) = \\ &\quad \texttt{roundupdPC}(\texttt{delaySN}_j - \mathit{delay} + \mathit{delaySN50}_j) \quad \wedge \\ &\qquad \mathit{delaySN50constr}_j = \texttt{dPC} - \texttt{rPerCoq}_j \mod \texttt{dPC} - \\ &\qquad (\texttt{delaySN}_j - \mathit{delay} + (\texttt{dPC} - 1)) \mod \texttt{dPC}) \\ &\vee \\ &(\texttt{roundupdPC}(\texttt{delaySN}_j - \mathit{delay}) < \\ &\quad \texttt{roundupdPC}(\texttt{delaySN}_j - \mathit{delay} + \mathit{delaySN50}_j) \\ &\quad \wedge \\ &\qquad (((\texttt{delaySN}_j - \mathit{delay}) \mod \texttt{dPC} = 0 \quad \wedge \\ &\qquad\quad \texttt{rPerCoq}_j \mod \texttt{dPC} = 0 \wedge \mathit{delaySN50constr}_j = 1) \vee \\ &\qquad ((\texttt{delaySN}_j - \mathit{delay}) \mod \texttt{dPC} = 0 \wedge \\ &\qquad\quad \texttt{rPerCoq}_j \mod \texttt{dPC} > 0 \quad \wedge \\ &\qquad\quad \mathit{delaySN50constr}_j = 1 + \texttt{dPC} - \texttt{rPerCoq}_j \mod \texttt{dPC}) \vee \\ &\qquad ((\texttt{delaySN}_j - \mathit{delay}) \mod \texttt{dPC} > 0 \wedge \texttt{rPerCoq}_j \mod \texttt{dPC} = 0 \quad \wedge \\ &\qquad\quad \mathit{delaySN50constr}_j = \texttt{dPC} - (\texttt{delaySN}_j - \mathit{delay}) \mod \texttt{dPC} + 1) \vee \\ &\qquad ((\texttt{delaySN}_j - \mathit{delay}) \mod \texttt{dPC} > 0 \quad \wedge \\ &\qquad\quad \texttt{rPerCoq}_j \mod \texttt{dPC} > 0 \wedge \texttt{rPerCoq}_j \mod \texttt{dPC} \leq \\ &\qquad\qquad \texttt{dPC} - (\texttt{delaySN}_j - \mathit{delay}) \mod \texttt{dPC} \quad \wedge \\ &\qquad\quad \mathit{delaySN50constr}_j = \texttt{dPC} - (\texttt{delaySN}_j - \mathit{delay}) \mod \texttt{dPC} + 1) \vee \\ &\qquad ((\texttt{delaySN}_j - \mathit{delay}) \mod \texttt{dPC} > 0 \quad \wedge \\ &\qquad\quad \texttt{rPerCoq}_j \mod \texttt{dPC} > 0 \wedge \texttt{rPerCoq}_j \mod \texttt{dPC} > \\ &\qquad\qquad \texttt{dPC} - (\texttt{delaySN}_j - \mathit{delay}) \mod \texttt{dPC} \quad \wedge \\ &\qquad\quad \mathit{delaySN50constr}_j = \texttt{dPC} - (\texttt{delaySN}_j - \mathit{delay}) \mod \texttt{dPC} + \\ &\qquad\qquad \texttt{dPC} - \texttt{rPerCoq}_j \mod \texttt{dPC} + 1)) \qquad (10)\end{aligned}$$

Constraints (8)–(10) contain decision variables *delayPN50* and *delaySN50*. Our goal is to replace these variables by *delayPN50constr* and *delaySN50constr* in our MiniZinc constraint model, thus *delayPN50* and *delaySN50* should be eliminated. For example, constraint (8) will be replaced by

$$(delayPN50constr_i = 0 \vee ((\texttt{delayPN}_i - delay) \mod \texttt{dPC} = 0 \wedge \\ delayPN50constr_i = 1) \vee (\texttt{delayPN}_i - delay) \mod \texttt{dPC} > 0 \wedge \\ delayPN50constr_i = \texttt{dPC} - ((\texttt{delayPN}_i - delay) \mod \texttt{dPC}) + 1), \quad (11)$$

we add this constraint to the model and we replace the variable *delayPN50* by *delayPN50constr* in (1)–(5).

6 Proofs in Coq

We made several assumptions that are formalised in Coq as axioms.[1] Constraint (12) is based on 3GPP standard.

$$\texttt{dPC} \geq 320 \quad (12)$$

Constraint (13) is based on the property that the user equipment can read first paging messages with *delay* less than dPC and we assume that radio base station can receive first notification message with delay less than 50 ms.

$$0 \leq delay < \texttt{dPC} + 50 \quad (13)$$

Constraints (14), (15) and (16) are required by proofs in Coq, which use lemmas of natural arithmetics.

$$\texttt{rPer}_j > 2 \cdot \texttt{dPC} \quad (14)$$

$$\texttt{delayPN}_i > delay \quad (15)$$

$$\texttt{delaySN}_j > delay \quad (16)$$

$\texttt{delayPN}_1$ and $\texttt{delaySN}_1$ can be equal to 0, but we do not consider this case, since then we set $delayPN50_1$ and $delaySN50_1$ to 0. Constraint (14) is a stronger version of assumption we made in [11]. We assumed in [11] that $\texttt{rPer}_j > \texttt{dPC}$, but we did not take in consideration message delays.

We started by proving correctness of the constraint that was manually derived for delay. However, while applying tactics in Coq, we found that some cases were missing in the constraint. The parameter rPer did not occur in formula for the delay. The constraints described in the previous section were derived by analysing the required assumptions after applying tactics.

We proved in Coq that the delays *delayPN50* and *delaySN50* can be replaced by *delayPN50constr* and *delaySN50constr* in the constraint model, that is the solutions for arrays of decision variables *PagPN* and *PagSN* are not changed after this replacement.

We split the proof into several lemmas and theorems, where we used Coq library for basic Peano arithmetic. We formulated different theorems for paging messages of primary and secondary notifications. Since constraints on timestamps of paging messages of secondary notifications are more complex, they require more lemmas to prove.

The proof consists of several steps.

[1] The Coq model is available at https://github.com/astra-uu-se/SEFM18.

1. We proved general properties of `roundupdPC`. For example, we prove that

$$\begin{array}{l}\forall n, d \in \mathbb{N},\\ (d \leq 50 \wedge \texttt{roundupdPC}(n) < \texttt{roundupdPC}(n+d)) \rightarrow\\ \texttt{roundupdPC}(n+d) = \texttt{roundupdPC}(n) + \texttt{dPC}\end{array}$$

2. We proved that $delayPN50constr_i \leq delayPN50_i$ and $delaySN50constr_j \leq delaySN50_j$, $1 \leq i \leq \texttt{nPrim}$, $1 \leq j \leq \texttt{nSec}$.
3. We proved that

$$\texttt{roundupdPC}(t + delayPN50constr_i) = \texttt{roundupdPC}(t + delayPN50_i),$$

   and

$$\texttt{roundupdPC}(t' + delaySN50constr_j) = \texttt{roundupdPC}(t' + delaySN50_j),$$

   where t, t' are expressions from constraint model.
4. Let $t' \mod \texttt{dPC} = 0$. We proved that
   - if $t' < (\texttt{delayPN}_i - delay) + delayPN50_i$ and $t' \geq (\texttt{delayPN}_i - delay)$ then $t' < (\texttt{delayPN}_i - delay) + delayPN50constr_i$.
   - if $t' < (\texttt{delaySN}_j - delay) + delaySN50_j$ and $t' \geq (\texttt{delaySN}_j - delay)$ then $t' < (\texttt{delaySN}_j - delay) + delaySN50constr_j$.
5. In constraint (10) $delaySN50constr_j$ depends on k. We showed that we do not need to have two or more values of $delaySN50constr_j$ with different values of k for the same $delaySN50_j$, $\texttt{delaySN}_j$ and $delay$. We proved that we can always choose the largest value of $delaySN50constr_j$.

From Step 2 we derived that

- $delayPN50constr_i \leq 50$,
- $delaySN50constr_j \leq 50$,
- if $t > (\texttt{delayPN}_i - delay) + delayPN50_i$, then $t > (\texttt{delayPN}_i - delay) + delayPN50constr_i$,
- if $t > (\texttt{delaySN}_j - delay) + delaySN50_j$, then $t > (\texttt{delaySN}_j - delay) + delaySN50constr_j$

Constraint (10) can be simplified by removing $2 \cdot \texttt{dPC}$, since we have expression $\texttt{rPerCoq}_j + 2 \cdot \texttt{dPC} = \texttt{rPer}_j \cdot k - 2 \cdot \texttt{dPC} + 2 \cdot \texttt{dPC}$. We add $2 \cdot \texttt{dPC}$ in order to be able to use lemmas of natural arithmetics in Coq. We have $\texttt{delaySN}_j - delay - r \leq 0$, but $\texttt{rPer}_j \cdot k - 2 \cdot \texttt{dPC} > 0$ by our assumption and $2 \cdot \texttt{dPC} + \texttt{delaySN}_j - delay - r > 0$.

7 Related Work

Because of the importance of network protocols there have been many case studies on the application of formal methods to the verification and study of network protocols. Some of the early work included the use of finite automata, Petri nets and symbolic execution to verify the absence of deadlock and liveness properties, see [19]. It is impossible here to give a complete survey of the field,

(see [17] for a recent survey) instead we will concentrate on the use of theorem provers based on type theory, such as Coq or Isabelle [16] to verify non-trivial properties of network protocols.

One advantage of using a theorem prover over a model checker is that it is easier to reason about infinite objects and hence prove properties that are satisfied by every possible run of a protocol. In [9] a novel use of co-inductive types, which correspond to infinite streams of data, was used to model and verify the alternating bit protocol. While [9] used a process calculus that characterises possible computation steps that can be performed in a protocol, the work in [5] uses an algebraic approach that captures when two processes are equivalent. In order to automate a hand-written proof of the alternating bit protocol, an encoding of a rich and widely used algebraic specification language for concurrent systems with data, μCRL [12] was formalised in Coq.

More recent work (see [3] and the references therein) on process calculi in Isabelle has resulted in generic framework, the Psi-Calculus, that captures many different process calculi. The resulting formalisation is over 32,000 lines of Isabelle code. In [7] an extension of the Psi-calculus was given to capture the broadcast of messages and applied to a non-trivial wireless sensor network protocol.

The use of interactive theorem proving for high-level constraint models in languages similar to MiniZinc has been considered in [4,8]. In [4], the authors show that almost all interesting properties of a constraints model, such as model equivalence, are undecidable in general for languages as expressive as MiniZinc. However, they illustrate that properties can be automatically verified when a restricted language is considered. In [8] interactive theorem proving was used to derive symmetry breaking constraints of models. The work in [4,8] considers the general problem of reasoning about any model, while we consider a specific case study where undecidability is not a problem.

All of the cited work so far has been concerned with protocols that do not have a time component. In our application the timing of messages is of crucial importance to the correct operation of the protocol. In [13] a real time protocol is verified using HOL (closely related to Isabelle [16]). Further resulting formalisation is then used in conjunction with the theorem analyse the qualitative soft real-time behaviour of the protocol. This is similar in spirit our derivation of message delays.

8 Conclusion

We used Coq to discover a formula for message delay in a 4G protocol and proved correctness of the formula.

We analyzed real logs from [11] with derived delays. However, there was an impact on performance of large generated log analysis. We still can analyze logs with large number of errors, but with a size smaller than in [11].

We found that Coq is a useful tool. It would be hard to do such proofs by hand and the tool helped to construct formula for delay. We want to emphasize

that interactive feature of Coq was very important, since the formula in the initial model was not correct. Proofs are about 6500 lines, but we believe that number of lines can be reduced by improving our use of tactics. Our proofs are based on the structure of the constraint model used in [11]. The proofs could also be done with Isabelle/HOL or PVS. In this formal development we rely on Coq features related to inductive types and first order logics and last but not least on Coq standard library (for modulo). We also used intensively the omega tactic to solve some arithmetic subgoals. Why3 [6] could be an alternative for our work, offering a large choice of automatic solvers and proof assistants. However it depends on the way these solvers support the modulo operator. A perspective is to apply this approach to other protocols with message delay to create more efficient constraint model. An interesting question to explore is how to convert automatically constraint models into Coq.

Acknowledgments. The second author was supported by Swedish Foundation for Strategic Research. The third author is partially support by the Swedish Research Council VR.

References

1. 3GPP. Public warning system (PWS) requirements. TS 22.268, 3rd Generation Partnership Project (3GPP). http://www.3gpp.org/ftp/Specs/html-info/22268.htm
2. Barras, B., Werner, B.: Coq in Coq. Technical report, INRIA-Rocquencourt (1997)
3. Bengtson, J., Parrow, J., Weber, T.: Psi-Calculi in Isabelle. J. Autom. Reasoning **56**(1), 1–47 (2016)
4. Bessiere, C., Hebrard, E., Katsirelos, G., Kiziltan, Z., Narodytska, N., Walsh, T.: Reasoning about Constraint Models. In: Pham, D.-N., Park, S.-B. (eds.) PRICAI 2014. LNCS (LNAI), vol. 8862, pp. 795–808. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13560-1_63
5. Bezem, M., Bol, R., Groote, J.F.: Formalizing process algebraic verifications in the calculus of constructions. Formal Aspects Comput. **9**(1), 1–48 (1997)
6. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Workshop on Intermediate Verification Languages (2011)
7. Borgström, J., Huang, S., Johansson, M., Raabjerg, P., Victor, B., Pohjola, J.Å., Parrow, J.: Broadcast psi-calculi with an application to wireless protocols. Softw. Syst. Model. **14**(1), 201–216 (2015)
8. Cadoli, M., Mancini, T.: Using a theorem prover for reasoning on constraint problems. Appl. Artif. Intell. **21**(4&5), 383–404 (2007)
9. Giménez, E.: An application of co-inductive types in Coq: verification of the alternating bit protocol. In: Berardi, S., Coppo, M. (eds.) TYPES 1995. LNCS, vol. 1158, pp. 135–152. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61780-9_67
10. Gonthier, G.: The four colour theorem: engineering of a formal proof. In: 8th Asian Symposium of Computer Mathematics, p. 333. ASCM (2007)
11. Grinchtein, O., Carlsson, M., Pearson, J.: A constraint optimisation model for analysis of telecommunication protocol logs. In: Blanchette, J.C., Kosmatov, N. (eds.) TAP 2015. LNCS, vol. 9154, pp. 137–154. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21215-9_9

12. Groote, J.F., Ponse, A.: The syntax and semantics of μCRL. In: Ponse, A., Verhoef, C., van Vlijmen, S.F.M. (eds.) Algebra of Communicating Processes. Workshops in Computing. Springer, London (1995). https://doi.org/10.1007/978-1-4471-2120-6_2
13. Hasan, O., Tahar, S.: Performance analysis and functional verification of the stop-and-wait protocol in HOL. J. Autom. Reason. **42**(1), 1–33 (2009)
14. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
15. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007)
16. Nipkow, T., Paulson, L.C., Wenzel, M. (eds.): Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
17. Qadir, J., Hasan, O.: Applying formal methods to networking: theory, techniques, and applications. IEEE Commun. Surv. Tutorials **17**(1), 256–291 (2015)
18. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Elsevier, New York (2006)
19. Sunshine, C.A.: Survey of protocol definition and verification techniques. SIGCOMM Comput. Commun. Rev. **8**(3), 35–41 (1978)
20. The Coq Development Team. The Coq proof assistant reference manual version 8.6 (2016)

Automated Validation of IoT Device Control Programs Through Domain-Specific Model Generation

Yunja Choi(✉)

School of Computer Science and Engineering,
Kyungpook National University, Daegu, South Korea
yuchoi76@knu.ac.kr

Abstract. The IoT is a networked system of physical devices controlled by embedded software whose validity is a pre-requisite to ensuring the correct behavior of the entire system. To automate the verification and validation process of such control software, this work constructs a validation model by composing pre-defined behavioral patterns of an operating system that is compliant with the OSEK/VDX international standard and models of application programs abstracted w.r.t. interactions with the underlying operating system. This validation model is used to perform property checking using the model checker SPIN to ensure that the behavior of the control program complies with the original intention of the program design. We automated the model generation process and applied it to 9 benchmark programs for the open source IoT OS Erika.

1 Introduction

Verification and Validation (V&V) of embedded control software has been an active research area, especially for software controlling safety-critical systems [10, 23]. The importance of rigorous and efficient V&V methods and tools cannot be over-emphasized in safety-critical domains. Especially in the era of IoT systems, which connect everything in our daily lives - from simple devices, such as sensors and actuators, to complex machines such as cars, drones, and home electronics -, the importance of the quality assurance for each device controller is increasing ever more. The bottom line is that the correct behavior of the whole IoT system cannot be assured without assuring each contributing device.

Numerous approaches have been suggested for comprehensive and rigorous verification of control software, represented by formal modeling and formal verification [3,14], model-based test generation [2,6,22], and model-driven code generation [15], with many visible achievements in practice. Nevertheless, existing approaches are not very suitable for IoT device control software because

This research was supported by Basic Science Research Program through National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2016R1D1A3B01011685) and the Ministry of Science, ICT (No. 2017M3C4A7068175).

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 254–268, 2018.
https://doi.org/10.1007/978-3-319-92970-5_16

of several reasons, including: (1) Existing approaches generally do not take the underlying operating systems into account, even though it is an inseparable part of the small-scale control software; (2) model construction for each device controller requires domain and modeling experts; and (3) rigorous and comprehensive verification typically requires formal verification techniques and experts in this field. Experts in these areas are quite rare in practice, and, even when they are available, it is time-consuming and impractical to apply manual modeling and verification to potentially hundreds of devices contributing to an IoT system.

This work proposes the first step towards quick and easy formal verification through automated model generation for device control software written in the C language. Given a control program and its configuration information, our approach auto-constructs a validation model by composing a set of service patterns of a standard operating system and the target control software. The service patterns defined in parameterized statemachines [5] are written in PROMELA, the input language of the model checker SPIN [17]. The control software written in C is embedded in the PROMELA application model, which interacts with the OS model only through API function calls. We introduce two major components in our model construction approach, the OS model written in PROMELA and the application model embedded in the PROMELA interaction model.

The validation model is a formal representation of the device controller focusing on the interactions between the control program and its underlying operating system while abstracting from the implementation details. This model is used to simulate the control software and to rigorously validate the correct behavior of the control program using model checking. As long as each device controller works on an operating system that is compliant with the same international standard, our approach fully automates the construction of the model from control software written in C and enables automated formal validation of the control logic using model checking. The suggested approach is applied to nine Erika benchmark programs [1] to perform comprehensive property verification as a means for validating the behavior of each control program w.r.t. the correct sequence of task executions.

The remainder of this paper is organized as follows. Section 2 explains the characteristics of the development of IoT device controllers and provides a brief overview of SPIN and PROMELA. Section 3 introduces our approach for configurable model construction. Section 4 shows the result of the case study where the suggested approach was applied to a set of benchmark programs of the Erika. We conclude with a brief summary of related work (Sect. 5) and a discussion (Sect. 6).

2 Background

Our approach is a domain-specific implementation of the pattern-based V&V framework introduced in [5], using the modeling language PROMELA and the model checker SPIN. This section provides a brief overview of our target domain and some basic knowledge about PROMELA and SPIN to help understand the essentials of the suggested approach.

2.1 Elements of Device Control Software

The control software of an IoT device is typically running on top of an operating system. A typical set of services commonly required by IoT devices includes services for task management, resource management, event management, alarm management, interrupt handling, and task scheduling. Contiki, RIOT, FreeRTOS, Zephyr, and Erika are representative examples of operating systems specialized for small-scale IoT devices.

One thing to note is that a control program is inseparable from its underlying operating system by construction. The software part of the device controller is produced by compiling a selected subset of operating system services together with the application program, where the selection of required services is determined by the configuration of each application program such as the numbers and types of tasks, alarms, and ISRs. Therefore, validation of such device controllers also requires considering configuration-dependent operating system services and the interaction behavior among the operating system and application programs. This is not a trivial task as the same control logic may behave differently in different system configurations.

2.2 Model Checker SPIN and C Code Embedding

SPIN [17] is one of the most widely used software model checkers. It is quite accessible to engineers as it uses the C-like modeling language PROMELA and its checking mechanism is based on explicit statespace search, providing numerous options for optimizing the speed of the search. In particular, it supports embedding of C program code directly into the model, making it possible to smoothly integrate application source code written in C with high-level models.

Processes and Message Passing in PROMELA. The syntax of PROMELA is similar to the C language, supporting most primitive data types plus arrays and structures in C; its semantics, however, is based on CSP [16]. Major constructs of PROMELA include `proctype`, which defines the type of a process, and `chan`, which defines a communication channel used to pass messages among processes. The following is an example of a process that communicates through the channel `api_ch` consisting of four message fields. The size of the channel is declared as five, meaning that up to five messages are buffered in the channel.

```
chan api_ch = [5] of {byte, mtype, byte, byte};
proctype OSEK_OS(){
        // wait StartOS message
        api_ch?[_,eval(StartOS),_,_]; api_ch?_,_,_,_;
        // start OS logic goes here
        api_ch!tid, RT, 0,0;
        ...
}
```

Processes use the channel to check whether a specific message arrives and to selectively receive the message by evaluating its value. Unlike C, each statement in PROMELA is executed if the evaluation of the statement is true, but is blocked otherwise until it becomes true. For example, `api_ch?[_,eval(StartOS),_,_]` is executed when the `api_ch` channel receives a message where the value of the second field of the message is equal to `StartOS`. PROMELA also supports nameless reception of messages, such as `api_ch?_,_,_,_`, for cases where specific values of the messages are not of interest.

C Code Embedding in PROMELA. PROMELA is a modeling language specialized for rigorous verification of communicating processes using SPIN. Though it is supposed to be easy to use in practice due to its C-like syntax, modeling before programming is not a common practice in the development of embedded software, especially in the development of control software for small devices. PROMELA provides a bypass for this issue by allowing embedding of program source code directly into PROMELA models, thus saving time and effort for model construction. There are two major constructs for C code embedding, `c_decl` and `c_code`. `c_decl` is for the declaration of types and variables that are used inside the `c_code` block. Statements in a `c_code` block are executed according to the control flow, but value changes are not traced by the model checker, unlike the values of variables in a PROMELA model. It is also possible to trace a specific variable inside a `c_code` block, using a `c_track` construct, meaning that we can control the level of abstraction of the model by tracing only those values that are relevant for the verification goal.

3 Configurable Model Construction in Promela

As shown in Fig. 1, our validation model is a SPIN-executable PROMELA model, which is a composition of an OS model constructed from a set of pre-defined OS patterns and an application model abstracted from C source code. This section introduces OS patterns defined in PROMELA and the interaction model between an OS model and an application model. The way application programs are embedded in the interaction model is also explained.

3.1 OS Patterns in PROMELA

A minimal set of OS patterns are modeled in PROMELA by referencing the patterns defined as parameterized statemachines in [5], including models of kernel variables, basic API functions, alarms and ISRs.

OS Kernel Variables and Basic Operations. The patterns for minimal OS services are designed for priority-based FIFO scheduling and resource priority-ceiling protocol as required by the OSEK/VDX international standard and many other IoT operating systems. To maintain information regarding static and

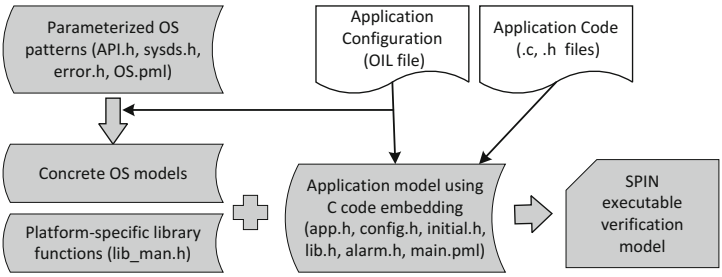


Fig. 1. PROMELA model generation

dynamic configurations of the system, the OS patterns include essential data structures as follows:

1. A priority queue: A priority queue is defined as a two-dimensional array whose elements are $<identifier, priority>$ of a task.
2. Static configuration of tasks: An array of static information of tasks whose elements consist of initial task priority and Boolean values indicating whether a task is preemptable, auto-start, or extended.
3. Dynamic task information: An array of dynamic task information including the number of current activations and the dynamic priority of a task.
4. Resource table: A two-dimensional array that maps a resource to a task that allocates the resource.
5. Event table: A two-dimensional array that maps a task to events that the task is setting or waiting for.

We also modeled basic operations common to all operating systems that use priority queues, such as pushing tasks to the priority queue and getting the information of a task from the queue.

Basic API Functions. An operating system provides a set of API functions to be used by application programs to fulfill their specific tasks. In this respect, we define an OS as a set of basic API functions that are modeled as inline functions in PROMELA.

Table 1 shows the list of major API functions that require task scheduling and corresponding headers of inline functions specified in PROMELA[1]. The behavior of each API function is modeled by referencing the formal models defined in parameterized statemachines in our previous work [5], where the caller of the API function t_c is specified explicitly. For example, `inline activate_task(param0, param1)` shown in the left part of Fig. 2 models the activation behavior of task `param1` requested by task `param0`. It first checks whether the current number of activations of task `param1` is within the maximum number of activations and

[1] The names of these API functions may differ among OSes, but most OSes specialized for IoT devices offer the same functionalities under different names.

Table 1. API functions in PROMELA

API function	PROMELA	Usage
ActivateTask(t)	inline activate_task(t_c, t)	t_c requests the activation of task t
TerminateTask()	inline terminate_task(t_c)	t_c requests the termination of itself
ChainTask(t)	inline chain_task(t_c, t)	t_c requests the activation of task t and the termination of itself
GetResource(r)	inline get_resource(t_c, r)	t_c requests an allocation of resource r
ReleaseResource(r)	inline release_resource(t_c, r)	t_c requests deallocation of resource r
WaitEvent(e)	inline wait_event(t_c, e)	t_c requests waiting for event e
SetEvent(t, e)	inline set_event(t_c, t, e)	t_c requests setting an event e for task t
Schedule()	inline schedulet(t_c)	t_c requests rescheduling

pushes the task into the priority queue, setting the state of the task *Ready*. It then checks whether rescheduling is required, i.e., whether the newly activated task has higher priority than the currently running task `param0`. If so, it preempts the currently running task by pushing it into the priority queue and gets a task with the highest priority from the priority queue.

Alarms and ISRs. An alarm used in embedded software is declared in the configuration file, but details of its behavior can be specified either in the configuration file as an autostart alarm or in the application source code using API function calls, such as `SetAbsAlarm` or `SetRelAlarm`. Below is a typical configuration specification for an alarm written in the OIL configuration language. It specifies that the `testalarm` is an auto-start alarm, meaning that it automatically starts when the OS starts, and activates the task `my_periodic_task` at every 2000 ms. If the `AUTOSTART` flag is `FALSE`, then the alarm is not started until the application program calls `SetAbsAlarm` or `SetRelAlarm` with the parameters specifying the `ALARMTIME` and `CYCLETIME`.

```
ALARM testalarm {
        COUNTER = SystemCounter;
        ACTION = ACTIVATETASK { TASK = my_periodic_task; };
        AUTOSTART = TRUE { APPMODE = std;
        ALARMTIME = 100; CYCLETIME = 2000; };
};
```

An alarm is used to perform a task or to set events periodically, but does not contain control logic on its own. It does not have its own priority, nor is it scheduled by the scheduler, and thus it does not go through the priority queue

```
inline activate_task(param0, param1){
  atomic{
    if
    :: task_dyn_info[param1].act_cnt <
      task_static_info[param1].max_act_cnt ->
         task_dyn_info[param1].act_cnt++;
         prio = task_static_info[param1].prio
         push_task_into_readyQ(param1,prio,0);
         task_state[param1] = Ready;
    :: else -> e_code = E_OS_LIMIT;
    fi;

    if
    :: task_dyn_info[run_tid].dyn_prio < max_prio
       ->  prio = task_static_info[run_tid].prio;
           push_task_into_readyQ(run_tid,prio,1);
           task_state[run_tid] = Ready;
           get_task_from_readyQ(tid, prio);
           task_state[tid] = Running;
           run_tid = tid;
    :: else -> skip;
    fi;
  }
}
```

```
1 : proctype Alarm(){
2 :   //local variables here
3 :   init_state:
4 :       api_ch?[_, eval(SetRelAlarm),_,_];
5 :       api_ch?ctid,_,_,cycle;
6 :       api_ch!tid, RT, ctid, 0;
7:         i =0;
8 :   timer:
9 :        if
10:        :: i == cycle ->
                    api_ch!run_tid,ActivateTask,2,0;
11:                 api_ch?[_, eval(RT), eval(run_tid),_];
12:                 api_ch?_,_,_,_;
13:                  i=0;
14:       :: i < cycle -> i++;
15:       :: else -> i=0;
16:       fi;
17:  goto timer;
}
```

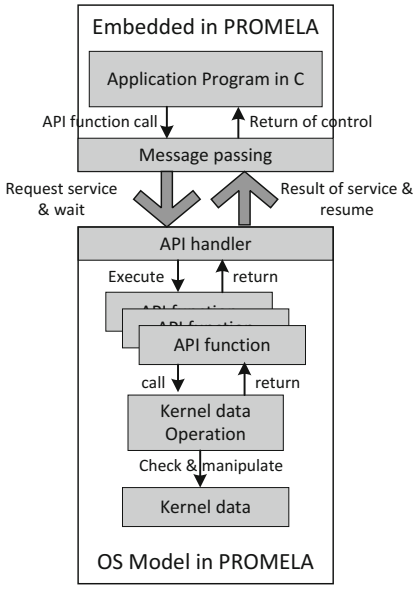
Fig. 2. Patterns in PROMELA: activate_task inline function and Alarm proctype

to perform its work. Therefore, we modeled the alarm as an independent process rather than a kind of task that needs to be maintained and scheduled by the task manager and the scheduler. The right part of Fig. 2 is a pre-defined model of an alarm in PROMELA that is not autostart. Once instantiated, the alarm waits for `SetRelAlarm`, which sets the timer and the cycle (lines 4–6). Lines 8–16 are for modeling a timer and the action performed at each end of the cycle. The model is auto-generated depending on the configuration of the alarm, which is parameterized by whether it is an autostart alarm or not; lines 3–6 are included in the model only if the alarm is not autostart. The action to be performed by the alarm at each end of the cycle (e.g. `ActivateTask` in line 10) is also parameterized in the model generation process. The value of the parameter is determined by the system configuration.

On the other hand, the model for Interrupt Service Routines (ISRs) is defined as a kind of task, as ISRs and tasks share similar characteristics: They are allowed to contain application logic, have their own priorities, and are scheduled. A major difference between an ISR and a task is that an ISR is triggered by external signals while a task is executed according to the task execution sequence determined by the scheduler. There are minor differences between them, e.g., ISRs are restricted to calling certain API functions. The left side of Fig. 4 is an example code of an ISR. Models of tasks and ISRs are not predefined, but auto-constructed from the source code by using code embedding. Details will be explained in the subsequent sections.

3.2 Interaction Model

OS kernel variables, basic API functions, and basic system operations are pre-defined and selectively composed to construct an OS model depending on the choice of system configuration. This OS model is again composed with application code written in C after it is embedded into the PROMELA interaction model. The left part of Fig. 3 shows an overview of the model representing the interaction between the OS model and an application program.




```
proctype API_handler(){
  // declaration of local variables
  Off:
   api_ch?[_,eval(StartOS),_,_]; api_ch?_,_,_,_;
   goto On;
  On:
   if
   :: d_step{ api_ch?[_,eval(ActivateTask),_,_]
      -> api_ch?param0,_,param1,_; }
         activate_task(param0,param1);
         api_ch!0,RT,param0,0;
   :: d_step{ api_ch?[_,eval(Schedule),_,_]
      -> api_ch?param0,_,_,_; }
         schedule(param0);
         api_ch!0,RT,param0,0;
   :: ...
   fi;
  goto On;
}
```

Fig. 3. IModel of interaction between application code and OS model

The API handler in the PROMELA model is an independent process that receives requests from other processes (mainly from the application program) and calls the inline API function corresponding to the request. The right part of Fig. 3 is a skeleton of the API handler in PROMELA: It waits for messages through the message channel `api_ch`. Once a message arrives, it checks whether the message corresponds to the API function it has been waiting for; if so, it removes the message from the channel, and performs the specified services. For example, once the OS starts, the API handler is in the `On` state and waits for API function calls such as `ActivateTask` or `Schedule`. If it receives a message for `ActivateTask`, it executes the pre-defined inline function `activate_task`, returns the control to the application program, and goes back to the `On` state waiting for other messages.

Application programs are embedded into the PROMELA interaction model by wrapping the application logic with `c_code` blocks. Figure 4 is an example

of embedding an application program source code into the interaction model. The left side of the figure shows an interrupt handler named `TEST_IRQ2`. By constructing a CFG (Control Flow Graph), the C code is analyzed w.r.t. the call to API functions in order to identify code blocks that do not contain API function calls. The code blocks are embedded into the PROMELA interaction model by enclosing them within `c_code`. Parts that call API functions are converted into message passing statements in PROMELA.

We note that the model checker SPIN does not trace the execution of a `c_code` block unless it is explicitly specified to be traced. Therefore, code blocks wrapped with `c_code` are not accounted for in terms of verification cost.

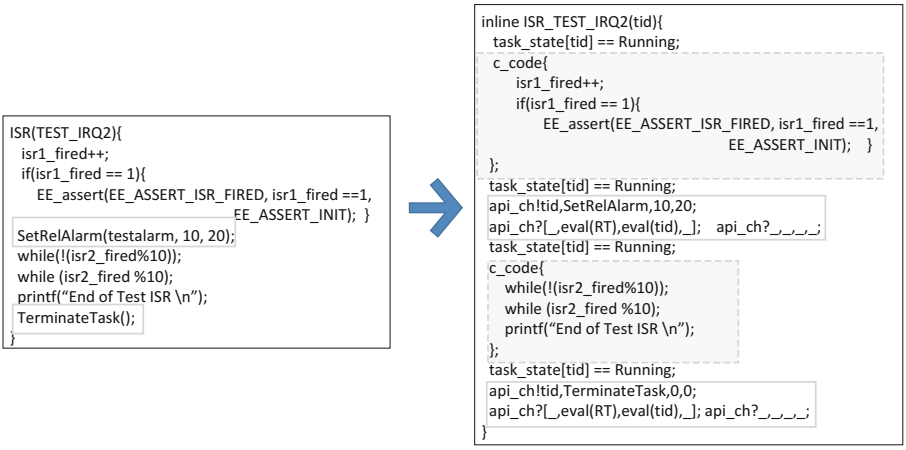


Fig. 4. An example of application code embedding

3.3 CFG Annotation for C Code Embedding

In order to embed C code into the PROMELA interaction model, we perform an analysis on the CFG of the program source code in order to identify code blocks that do not contain API function calls and to annotate the interaction points with PROMELA message passing statements. We first merge consecutive nodes that do not contain API function calls into one state block, which becomes the unit of code embedding.

Figure 5 illustrates our CFG annotation method for each unit function where each node that calls an API function is annotated as an interaction point and the maximal code blocks that do not contain API function calls are annotated as `c_code` blocks. If a task or a user-defined function does not contain any API function call, the entire function can be annotated with `c_code`. Nevertheless, as an application program may consist of several tasks, ISRs, and user-defined functions, with non-trivial call structure, we first analyze the call graph to check whether a function containing a call to an API-function exists in the call hierarchy before annotating each unit function. Below is the overall annotation process.

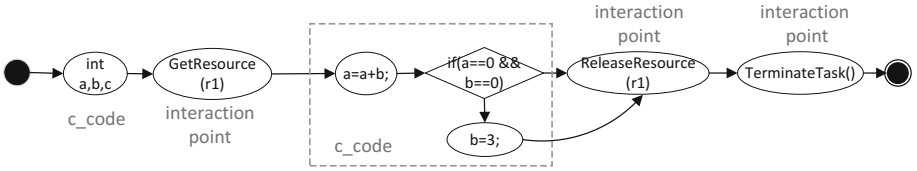


Fig. 5. CFG annotation for C code embedding

1. Construct a CFG for each task, ISR, and the main function. Store them in a set of CFGs, say `CFGset`.
2. For each task/ISR/main, perform call graph analysis to identify all functions (callees) directly or indirectly called by the task/ISR/main and store their CFGs in `CFGset`. In this process, the head node of each callee is annotated with `APIcall` if it contains an API function call. If a callee is annotated with `APIcall`, all direct/indirect callers of the callee are also annotated in the same way.
3. For each CFG in `CFGset`, perform annotation for C code embedding if the head node of the CFG is annotated with `APIcall`; otherwise, embed the entire function.

After the CFG annotation, each unit function is translated into a PROMELA model by converting statements annotated with `c_code` and `interaction point` into `c_code` blocks and PROMELA message passing statements, respectively. The right part of Fig. 4 is an example of the conversion. `c_code` blocks are embedded as they have no syntactic changes. The interaction points, however, are translated into PROMELA interaction statements for sending requests and receiving the result of the request. We also note that the interaction statements are preceded by the checks for the current state of the caller task or ISR requesting the service. The PROMELA statement `task_state[tid]==Running` is to ensure that the task with the identifier *tid* is currently in the *Running* state in order to call an API service. If this is not the case, the task is blocked until it gets executed.

4 Case Study: Applications to Erika Programs

We applied our approach to nine benchmark programs of the Erika OS for the x86 platform. Erika [1] is an open-source free RTOS compliant with the OSEK/VDX standard and the MISRA 2004 coding standard. It supports hard real time with fixed-priority scheduling, immediate priority ceiling, as well as Earliest Deadline First scheduling. As our patterns are modeled based on the basis of the OSEK/VDX standard, Erika applications can be considered a good starting point for validating our approach.

Table 2 shows some characteristics of these nine programs. From left to right, each column represents the name of the program, the lines of source code (excluding comments, library functions, and configuration files), the numbers of tasks,

Table 2. Erika benchmark programs

Name	LOC	T	A	I	R	Characteristics
EEtest00	36	1	0	0	0	Used for testing the priority-based scheduling of a task
EEtest01	57	2	0	0	0	Used for testing an application that activates a task from the main function
EEtest02	82	2	1	0	0	Sets an alarm in the main function and activates a task in the alarm handler
EEtest03	106	3	0	1	0	Activates a task in the main function and the task triggers an interrupt whose handler activates another task
EEtest04	112	3	1	1	0	Activates a task in the main function that sets a periodic alarm. The alarm activates another task
EEtest05	140	3	1	1	0	Activates a task in the main function that triggers an interrupt in the presence of a periodic alarm
EEtest06	135	3	1	0	0	Uses nested user-defined functions in addition to typical tasks and ISRs
EEtest07	123	3	1	1	0	A task activates itself over the maximum number of activations allowed
EEtest08	145	3	1	1	1	Two tasks access the same resource

alarms, ISRs, and resources used in the programs, and the key characteristics of each program.

It is difficult to be sure how each program behaves, especially in the presence of periodic alarms and interrupts. Simulation has been a major means for ensuring the validity, but it is difficult to consider it comprehensive. For more comprehensive and automated validation of the behavior of each application program, we have identified a list of properties that specify the expected execution sequences of tasks in the given application program. Some properties are common to all application programs, such as properties for ensuring mutual exclusiveness of multiple tasks or deadlock-freeness. Some are application-specific, as each application program has its own design of task execution sequences. Below are shown some of the identified properties.

S1. Task1 and Task2 shall not run at the same time.
S2. Running tasks shall terminate in the end.
S3. The number of activations of a task shall not exceed the specified maximum number of activation counts of the task.
S4. Once tasks are activated, the control shall never return to the main function (EEtest05).
S5. Task1 always triggers the interrupt handled by its corresponding ISR (EEtest05).
S6. Activation of the ISR always triggers the periodic alarm that activates Task2 (EEtest05).

S1 to S3 are common properties that must be satisfied by any application programs. S4 to S6 are examples of application-specific properties. These properties are formally specified in LTL (Linear Time Logic), which can be checked using the model checker SPIN.

P1. [] !((task_state[i] == Running) && (task_state[j] == Running)) (for $i \neq j$)
P2. [] ((task_state[i] == Running) → <> (task_state[i] == Suspended)) ($\forall i$)
P3. [] (task_dyn_info[i].act_cnt ≤ task_static_info[i].max_act_cnt) ($\forall i$)
P4. [] (task_state[1] == Running → ! <> ((task_state[0] == Running)))
P5. [] (task_state[1] == Running → X <> (task_state[3] == Running))
P6. [] (task_state[3] == Running → (TRUE U (task_state[2] == Running)))

LTL is a propositional logic with a temporal operator, which mainly includes the [], <>, X, and U operators. It is recursively defined for any temporal logic ϕ: []ϕ means that ϕ is always true; <> ϕ means that ϕ is true sometime in the future; X ϕ means that ϕ is true in the next state. U is a binary operator: ψ U ϕ means that ψ is true until ϕ is true. For example, property `P2` says that "It is always the case that if $task_i$ is running, it will be suspended eventually", and property `P5` says that "It is always the case that if $task_1$ is running, from the next state, the ISR will run eventually". Here, task_state[0] and task_state[3] represent the states of the main function and the ISR, respectively.

About 8 to 10 properties were validated on each application program using the model checker SPIN. Validating each property took less than 400 MBytes and 6 s. Table 3 shows the validation cost when the six properties were checked for `EEtest05` using SPIN. From left to right, each column represents the name of the property, the memory consumed in mega bytes, the time needed for validation in seconds, the depth of the search performed by SPIN, and the numbers of states and transitions searched while the model checking was performed. All properties were validated using the model checking.

Table 3. Performance of validation using the model checker SPIN

Property	Memory	Time	Depth	States	Transitions
P1	317.69	3.20	2,793	1,696,494	2,613,928
P2	228.63	1.68	2,793	805,025	1,289,853
P3	260.66	2.32	2,793	1,236,665	1,876,919
P4	129.02	0.01	335	2,745	4,435
P5	228.63	1.77	2,793	805,025	1,289,853
P6	393.57	5.15	2,793	2,771,009	4,056,731

5 Related Work

There are a number of tools and approaches for model checking C programs [8,18,19,23]. CBMC [8] is one of the representative C code model checkers

with many applications and case studies [4,9,21]. It is based on bounded model checking techniques using SAT or SMT solvers, and can be applied directly to ANSI-C programs without performing abstractions or model extraction. However, it suffers from a couple of drawbacks when it is to be applied to IoT device controllers. First, its performance on multi-threaded programs is quite low, and IoT device controllers typically consist of multiple tasks. Second, it is difficult to handle the behaviors of operating systems as this requires handling of context switching and access to hardware memory. Modeling of context switching behavior is necessary to verify behaviors affected by the operating system, but how to model the context switch and how to combine this model with C source code are not trivial problems.

SPIN [17] is a more natural choice as it supports specification of multiple processes, but it requires modeling using PROMELA to apply the model checking engine. Applying SPIN model checking to C programs has been an active research issue [18,19,23], including tools for automatic model extraction from C source code [19,20]. In particular, reference [19] converts C programs into PROMELA models through code embedding. It is fully automated, but has several problems as it is designed as a general model extractor. For example, it is difficult to understand the extracted models as well as the verification result due to the changes and additions of variable names[2], and complex data structures and nested calls are not handled properly. As the model extraction focuses on application programs without considering the underlying operating system, how to integrate the extracted models with operating system models is also an issue.

There are other approaches for extracting models from C source code for model checking purposes [11–13,20], but they rarely consider interactions with operating systems in the model extraction process. Most of the approaches faithfully translate each statement of a C program into the target modeling language with minor abstractions, which is highly likely to cause a statespace explosion problem in model checking due to the implementation-specific characteristics of the C language. Some other approaches tried to address the issues related to concurrency and scheduling by modeling the operating system using C [7] or through translation into sequential programs [24,25]. The former approach is limited to the capability of CBMC, which is not efficient in dealing with loops and multiple threads; the latter approach heavily depends on language translation, leaving very little room for applying abstractions.

6 Discussion

We have presented a domain-specific model generation approach for rigorously validating device control programs. The proposed approach can be fully automated so that any control programs compliant with the international standard OSEK/VDX can be auto-translated into a validation model. The modular structure of the suggested framework also supports flexible adaptation for IoT operating systems that are not compliant with OSEK/VDX, because it is

[2] The names of the variables are changed to ensure the uniqueness of the names.

straightforward to change the models of API services without affecting other components of the interaction model.

The major difference between the suggested framework and existing approaches is two-fold: (1) Our approach is based on pre-defined service patterns of operating systems, which are used to generate configurable OS models; and (2) a specialized abstraction is applied through embedding C code into the interaction model, abstracting statements irrelevant to interactions with the underlying operating system, and thus, avoiding searches of uninteresting execution traces while model checking. The abstraction through C code embedding applies minimum modification of the C source code by identifying code blocks independent of API function calls and by annotating only interaction points. This approach not only saves verification cost but also provides high readability of the application model and verification result as the original code is rarely changed. The abstraction is sound w.r.t. the properties to be verified because interaction behavior is preserved before and after abstraction.

The current approach aggressively abstracts all implementation-specific details, focusing only on the sequence of task executions. Our next step is to extend the approach for checking the correctness of application programs through property-based data tracing. We are currently working on property-based data dependency analysis and automated tracing of variables in the dependency relation, a kind of property-based model refinement.

References

1. Erika realtime operating system. http://erika.tuxfamily.org/drupal/
2. Artho, C., Gros, Q., Rousset, G., Banzai, K., Ma, L., Kitamura, T., Hagiya, M., Tanabe, Y., Yamamoto, M.: Model-based API Testing of Apache ZooKeeper. In: Proceedings of 10th IEEE International Conference on Software Testing, Verification and Validation, pp. 288–298 (2017)
3. Berry, G.: Synchronous design and verification of critical embedded systems using SCADE and Esterel. In: 12th International Conference on Formal Methods for Industrial Critical Systems (2007)
4. Bucur, D., Kwiatowska, M.Z.: Poster abstract: software verification for TinyOS. In: 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (2010)
5. Choi, Y.: A configurable V&V framework using formal behavioral patterns for automotive control software. J. Syst. Softw. **137**, 563–579 (2018)
6. Choi, Y., Byun, T.: Constraint-based test generation for automotive operating systems. Softw. Syst. Model. **16**(1), 7–24 (2017)
7. Chung, Y., Kim, D., Choi, Y.: Modeling OSEK/VDX OS requirements in C. In: 24th Asia-Pacific Software Engineering Conference (2017)
8. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (2004)
9. Cordeiro, L., Fischer, B., Chen, H., Marques-Silva, J.: Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In: International Conference on Embedded Software and Systems (2009)

10. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. IEEE Trans. Softw. Eng. **38**(4), 957–974 (2012)
11. DuVarney, D.C., Purushothaman Iyer, S., Wolf, C.: A toolset for extracting models from C programs. In: 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems, pp. 260–275 (2002)
12. Gallardo, M.M., Joubert, C., Merino, P., Sanan, D.: A model-extraction approach to verifying concurrent C programs with CADP. Sci. Comput. Program. **77**, 375–392 (2012)
13. Gong, X., Ma, J., Li, Q., Zhang, J.: Automatic model building and verification of embedded software with UPPAAL. In: International Joint Conference of IEEE TrustCom/IEEE ICESS/FCST, pp. 1118–1124 (2011)
14. Graf, S.: OMEGA: correct development of real time and embedded systems. Softw. Syst. Model. **7**, 127–130 (2008)
15. Hili, N., Dingel, J., Beaulieu, A.: Modelling and code generation for real-time embedded systems with UML-RT and Papyrus-RT. In: IEEE 39th International Conference on Software Engineering Companion (2017)
16. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)
17. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Publishing Company, Reading (2003)
18. Holzmann, G.J., Joshi, R., Groce, A.: Model driven code checking. Automa. Softw. Eng. **15**, 283–297 (2008)
19. Holzmann, G.J., Ruys, T.C.: Effective bug hunting with Spin and Modex. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, p. 24. Springer, Heidelberg (2005). https://doi.org/10.1007/11537328_3
20. Ichii, M., Myojin, T., Nakagawa, Y.: A rule-based automated approach for extracting models from source code. In: 19th Working Conference on Reverse Engineering, pp. 308–317 (2012)
21. Kim, Y., Kim, M.: SAT-based bounded software model checking for embedded software: a case study. In: 21st Asia-Pacific Software Engineering Conference, pp. 737–748 (2014)
22. Matinnejad, R., Nejati, S., Briand, L.C., Bruckmann, T.: Automated test suite generation for time-continuous simulink models. In: IEEE/ACM 38th International Conference on Software Engineering, pp. 595–606 (2016)
23. Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmueller, T.: Incremental bounded model checking for embedded software. Formal Aspects Comput. **29**, 911–931 (2017)
24. Wu, X., Wen, Y., Chen, L., Dong, W., Wang, J.: Data race detection for interrupt-driven programs via bounded model checking. In: IEEE 7th International Conference on Software Security and Reliability Companion, pp. 204–210 (2013)
25. Zhang, H., Aoki, T., Chiba, Y.: Yes! you can use your model checker to verify OSEK/VDX applications. In: IEEE 8th International Conference on Software Testing, Verification, and Validation (2015)

Shape Analysis and Reuse

Graph-Based Shape Analysis Beyond Context-Freeness

Hannah Arndt, Christina Jansen, Christoph Matheja(✉), and Thomas Noll

RWTH Aachen University, Aachen, Germany
matheja@cs.rwth-aachen.de

Abstract. We develop a shape analysis for reasoning about *relational properties* of data structures. Both the concrete and the abstract domain are represented by hypergraphs. The analysis is parameterized by user-supplied *indexed graph grammars* to guide concretization and abstraction. This novel extension of context-free graph grammars is powerful enough to model complex data structures such as balanced binary trees with parent pointers, while preserving most desirable properties of context-free graph grammars.

One strength of our analysis is that no artifacts apart from grammars are required from the user; it thus offers a high degree of automation. We implemented our analysis and successfully applied it to various programs manipulating AVL trees, (doubly-linked) lists, and combinations of both.

1 Introduction

The aim of shape analysis is to support software verification by discovering precise abstractions of the data structures in a program's heap. For shape analyses to be effective, they need to track detailed information about the heap configurations arising during computations. Although recent shape analyses have become quite potent [1,6,8,13,20], discovering abstractions that go beyond structural shape properties remains far from fully solved. For example, this is the case when considering *balancedness properties* of data structures, such as the AVL property: A full binary tree is an AVL tree if and only if for each of its inner nodes, the difference between the heights of its two subtrees is -1, 0, or 1. In this setting, reasoning about constraints over lengths of paths or sizes of branches in a tree is required. However, as already noted in [8], inference of shape-numeric invariants "is especially challenging and is not particularly well explored."

We develop a shape analysis that is capable of inferring relational properties, such as balancedness, from a program and an intuitive data structure specification given by a *graph grammar*. Context-free graph grammars [14] have previously been successfully applied in shape analyses [15]. They are, however,

Matheja, C.—Supported by Deutsche Forschungsgemeinschaft (DFG) Grant NO 401/2-1.

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 271–286, 2018.
https://doi.org/10.1007/978-3-319-92970-5_17

not expressive enough to capture typical relational properties of data structures. Hence, we lift the concept of *indexed grammars* — a classical extension of context-free string grammars due to Aho [2] — to graph grammars. More concretely, we attach an *index*, i.e. a finite sequence of symbols, to each nonterminal. This information can then be accessed by the graph grammar to gain a fine-grained control over the applicable rules. For example, by using indices to represent the height of trees, a context-free graph grammar modeling binary trees can easily be lifted to a grammar representing *balanced* binary trees.

One strength of indexed graph grammars is that they offer an intuitive formalism for specifying data structures without requiring deep knowledge about relational properties. Furthermore, all key aspects of shape analysis (using the terminology of [20]) have natural correspondences in the theoretically well-understood domain of graph transformations: *Materialization*, an operation to partially concretize before performing a strong update of the heap, corresponds to the common notion of grammar derivations. *Concretization* then means exhaustively applying derivations. Conversely, *abstraction* (or canonicalization) coincides with applying inverse derivations as long as possible. In particular, effective versions of the above operations can be derived automatically from a grammar through existing normal forms [17]. Finally, checking for *subsumption* between two abstract states is an instance of the language inclusion problem for graph grammars. While this problem is undecidable in general [5], we present a fragment of indexed graph grammars with a decidable language inclusion problem that is well-suited for shape analysis.

We implemented our shape analysis and successfully verified Java programs manipulating AVL trees, (doubly-linked) lists and combinations of both.

Missing proofs as well as supplementary material to formalization and implementation are found in an extended version of this paper [4].

2 Informal Example

Our analysis is a standard forward abstract interpretation [11] that approximates for each program location the set of reachable memory states. It thus applies an abstract program semantics to elements of an abstract domain capturing the resulting sets until a fixed point is reached. The analysis is parameterized by a user-supplied *indexed hyperedge replacement grammar*: For any given grammar, we automatically derive an abstract program semantics from the concrete semantics of a programming language. Moreover, we obtain suitable abstraction and concretization functions. In this section we take a brief tour through the essentials of our approach by means of an example.

Example Program. We consider a procedure `searchAndSwap` (see Fig. 1) that takes an AVL tree `n` with back pointers and an integer value `key`. It consists of two phases: First, it performs a binary search in order to find a node in the tree with the given key (l. 9). If such a node is found, it moves back to the root of the tree (l. 13). However, before moving up one level in the tree, the procedure swaps the two subtrees of the current node (l. 12).

```
1  class AVLTree {
2    AVLTree left;
3    AVLTree right;
4    AVLTree parent;
5    int key;
6    // ...
7  }
```

```
8  void searchAndSwap(AVLTree n, int key) {
9    n = binarySearch(n, key);
10   while (n != null && n.parent != null) {
11     // swap subtrees of n
12     AVLTree t = n.left; n.left = n.right; n.right = t;
13     t = null; n = n.parent;
14   }
15 }
```

Fig. 1. Essential fields of class `AVLTree` and example code.

Abstract Domain. We assume a storeless model that is agnostic of concrete memory addresses. Memory states are then naturally modeled as graphs — more precisely *indexed heap configurations* (IHC) (Sect. 3). That is, an edge may be connected to an arbitrary number of nodes and is additionally labeled with an index that indicates, for instance, the height of a tree. Consider the IHC depicted in Fig. 2: A *node* (drawn as a circle) either represents an object or a literal, such as `null`, `true`, `false`, etc. The black circle denotes the special location `null`.[1] Pointers between objects are drawn as directed edges between two nodes that are drawn to indicate the corresponding field of its source object (`left` (dashed), `right` (dotted), and `parent` (solid) for AVL trees). For example, the `parent` pointer of the topmost node in Fig. 2 points to `null`. Furthermore, IHCs contain *program variables* and *nonterminal edges*. Program variables are drawn as diamonds that are labeled with the variable name and are attached to the unique node representing the value of the variable. Hence, variable n points to the rightmost node in Fig. 2. Nonterminal edges model a set of abstracted heap shapes, such as linked lists or balanced trees. They are drawn as gray boxes and attached to one or more nodes. Figure 2 contains two of these edges. Their label, B, indicates that both model a set of balanced binary trees. Further, their *indices*, X and sX, denote that they model balanced binary trees of height X and $X+1$, respectively, where X stands for an arbitrary non-negative value. Hence, the IHC in Fig. 2 models the set of all balanced binary trees with back pointers in which the height of the right subtree of the root is the height of its left subtree plus one. Moreover, variable n points to the right child of the root.

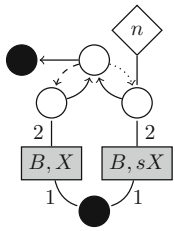


Fig. 2. An IHC

Abstraction and Concretization. The set of heaps described by an IHC is determined by an *indexed hyperedge replacement grammar* whose rules map nonterminal edges to an IHC. An example of a rule is provided in Fig. 3 (inside the gray box; above step (1)). Its left-hand side is $(B, s\nu)$, where ν is a variable. The rule allows to replace any edge that is labeled with B and whose index

[1] We often draw multiple black circles, but they all correspond to the same location.

starts with an s by the IHC below. In that case, variable ν is substituted by the remainder of the index of the replaced hyperedge. The IHC on the rule's right-hand side contains two *external nodes* (labeled 1 and 2) that indicate how two IHCs are glued together when replacing a hyperedge (Sect. 3).

Example Execution. Let us assume we are given a suitable grammar in which nonterminal B represents balanced binary trees and index sX stands for a height of $X+1$. We consider one execution sequence in detail. The individual execution steps are illustrated in Figs. 3, 4, and 5, respectively. Notice that the full analysis explores all abstract executions.

Step (1). Starting with the leftmost IHC in Fig. 3, we first execute a binary search (Fig. 1, l. 9). Assuming that the searched key is not at the root, we move to the children of `n`. Since these are currently hidden in the hyperedge labeled with (B, sX), we apply *materialization* [21] (partial concretization). For our analysis, materialization corresponds to forward derivations using the supplied graph grammar, i.e. we replace an edge by an IHC according to a rule of the grammar. Here, we used the rule above step (1) in Fig. 3. To apply this rule, we first remove the original hyperedge labeled (B, sX). After that we paste the graph belonging to the rule into the original graph. Finally, we identify the nodes originally attached to the removed hyperedge with the external nodes of the rule (as indicated by gray dashed and dotted lines in Fig. 3).

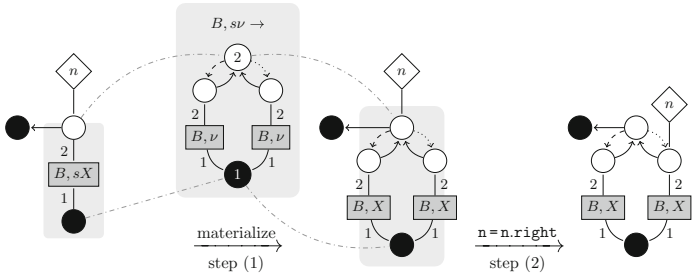


Fig. 3. Materialization and a possible execution of the binary search.

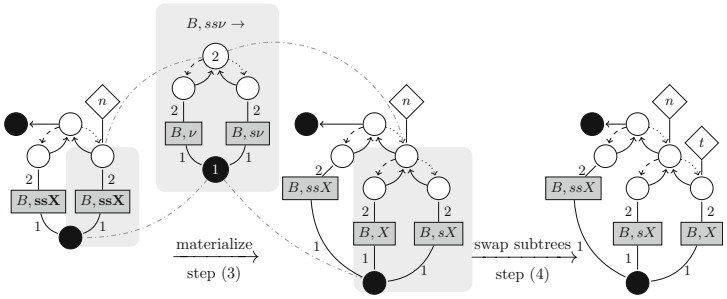


Fig. 4. Index materialization and swapping subtrees.

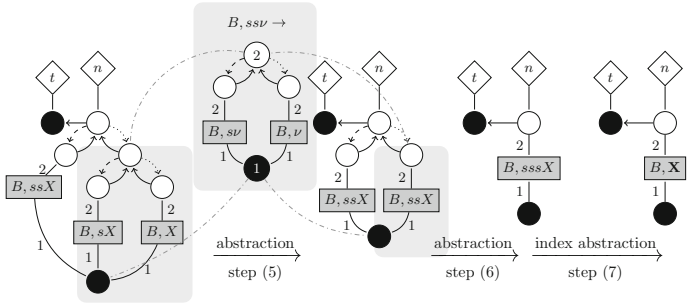


Fig. 5. Graph-based abstraction and index abstraction.

Step (2). After materialization, executing one step of the concrete program semantics amounts to a simple graph transformation (moving variable `n` to a child). To keep the example small, assume the binary search has already explored the left subtree without finding the key. It thus returned to the root and the next step is to move variable `n` to its right child. That is, we execute `n = n.right`. This leads to the rightmost graph depicted in Fig. 3. In our example execution, we assume `n` now carries the searched key, i.e. `n.key` equals `key`. Hence, the binary search returns the current position of `n` and we move to the `while`-loop of our example program (Fig. 1, l. 10). Since neither variable `n` is attached to `null` nor its `parent` pointer points to `null`, we enter the loop.

Step (3). Before we can climb up the tree to the root again, we have to swap the subtrees of `n` (Fig. 1, l. 12). Again, these are hidden in a hyperedge labeled with (B, X), i.e. we have to materialize again. As part of the example execution, we apply the rule in Fig. 4 (above step (3)). However, this rule requires the index of a hyperedge to be of the form $ss\nu$. Intuitively, this means the rule models balanced trees of height at least two. Since X is a placeholder for trees of arbitrary height, we apply *index materialization* to the IHC first. That is, we replace X by ssX in all hyperedges[2] and move to the leftmost hypergraph in Fig. 4. After that, we apply materialization as illustrated in the third step.

Step (4). We apply the concrete semantics to execute a sequence of assignments in order to swap the left and right subtree of `n` (Fig. 1, l. 12). This results in the rightmost IHC of Fig. 4, in which variable `t` has not been set to `null` yet. After executing the remaining two assignments, i.e. `t = null` and `n = n.parent`, we end up in the leftmost IHC in Fig. 5.

Notice that both the abstract semantics as well as materialization are derived automatically from the grammar and the concrete program semantics (Sects. 3 and 4). In particular, materialization corresponds to forward derivations using the grammar. Analogously, the abstraction function corresponds to applying backward derivations. Each occurrence of an IHC used as the right-hand side of a grammar rule is replaced by a hyperedge labeled with the rule's left-hand side.

[2] Again, note that we consider a single execution path in this example. The full analysis also explores the cases in which X is substituted by z and sz.

Step (5). After executing `n = n.parent` (Fig. 1, l. 13), abstracted is performed before moving on to the next loop iteration. We abstract using a rule symmetric to the one applied in step (3) for materialization. This corresponds to first detecting the IHC in the rule as a subgraph of the given IHC. This subgraph is deleted except for those nodes identified with the external nodes (labeled by numbers) of the rule graph (see gray dash-dotted lines in Fig. 5). Then a hyperedge attached to the latter nodes is added to the remaining IHC.

Step (6). The IHC obtained after step (5) can be further abstracted. This time, we employ the rule that has been applied for materialization first (Fig. 3, above step (1)). The resulting graph is found in Fig. 5 next to step (6). Note that the indices of both hyperedges to be abstracted are ssX whereas the rule used for abstraction contains hyperedges with indices ν. The variable ν is used as a placeholder to restore the original indices after the replacement. The resulting hypergraph (Fig. 5 following step (6)) contains a single hyperedge labeled $(B, sssX)$. Hence, the result of our example execution is a balanced binary tree (of height at least three) again.

Step (7). As a final operation, we apply the converse of index materialization in step (3): index abstraction. For this purpose, we replace $sssX$ by X, i.e. we generalize from trees of height at least three to trees of arbitrary height. Proceeding with the analysis, we evaluate the loop guard (Fig. 1, l. 10) to `false`, because `n.parent` equals `null`. Hence, the analysis terminates this branch of its execution with a final hypergraph that covers the initial one. The problem of checking whether a hypergraph covers another one is addressed in Sect. 5.

3 Program States and Indexed Grammars

As outlined in Sect. 2, it is intuitive to model heaps as graphs. In this section, we formalize heap configurations as a model for program states and their semantics in terms of a graph grammar. These grammars guide concretization and abstraction in our analysis, which is presented subsequently in Sect. 4.

3.1 Program States

To set the stage for our analysis, we consider program states to consist of a heap and a stack. We assume the heap to contain records with a finite number of *reference fields* that are collected in **Fields**. Apart from the heap, a program state is equipped with a *stack* mapping program variables in **Var** to records.

Furthermore, our abstract domain equips graphs with *nonterminal hyperedges* that act as abstract placeholders for sets of graphs, e.g. all (balanced) binary trees. These hyperedges are labeled with a *nonterminal* taken from a finite set N and an *index* taken from a finite set I, respectively. Throughout this paper, we fix a set $\mathbf{Types} = \mathbf{Fields} \cup \mathbf{Var} \cup N$. Every element of **Types** is ranked by a function $rank : \mathbf{Types} \to \mathbb{N}$, where fields always have rank two, i.e. $rank(\mathbf{Fields}) = \{2\}$ and variables always have rank one, i.e. $rank(\mathbf{Var}) = \{1\}$, respectively. Program states are then formally modeled as follows:

Definition 1. *An* indexed heap configuration *(IHC for short) is defined as a tuple* $H = (V, E, lab, att, ind, ext)$*, where*

- V *and* E *are finite sets of* nodes *and* hyperedges*, respectively,*
- $lab : E \to \mathbf{Types}$ *is a* hyperedge labeling function*,*
- $att : E \to V^*$ *maps each edge to a sequence of* attached nodes *that respects the rank of hyperedge labels, i.e. for all* $e \in E$*, we have* $rank(lab(e)) = |att(e)|$*.*
- $ind : E \to I^+$ *assigns a non-empty* index sequence *to each edge in* E*, and*
- $ext \in V^+$ *is a repetition-free sequence of* external nodes.[3]

Throughout this paper, we do not distinguish between the terms graph and hypergraph nor between edge and hyperedge. Furthermore, we refer to the components of a graph H by V_H, E_H, etc. If an edge e is attached to exactly two nodes, say $att(e) = uv$, we interpret e as a directed edge from node u to node v. Notice that all graphs in Sect. 2 are examples of IHCs.

To simplify the technical development, we impose a few sanity conditions on IHCs: We require that (1) every variable $x \in \mathbf{Var}$ occurs at most once in H and (2) for every field $f \in \mathbf{Fields}$ every node has at most one outgoing edge e labeled with f (recall that $rank(f) = 2$). The special location `null` is treated as a global variable. Hence, we assume a unique node $v_{\texttt{null}}$ representing `null` which is the first external node and the first node attached to every nonterminal edge.[4]

3.2 Indexed Grammars

The semantics of edges labeled with a nonterminal, is specified by an indexed graph grammar — an extension of context-free graph grammars. As it is common in graph rewriting, we do not distinguish between isomorphic graphs. Thus, *all sets of graphs in this paper are to be understood up to isomorphism.*

Definition 2. *Let* ν *be a dedicated* index variable *and* $I' = I \cup \{\nu\}$ *be the set of index symbols. An* indexed hyperedge replacement grammar *(IG) is a finite set of rules* G *of the form* $X, \sigma \to H$ *mapping a nonterminal* $X \in N$ *and an index* $\sigma \in I^*(I \cup \{\nu\})$ *to an IHC* H *such that* $rank(X) = |ext_H|$*. Moreover, if* σ *does not contain the variable* ν *then* H *does not contain* ν *either, i.e.* $ind_H(E_H) \subseteq I^+$*.*

Example 1. Figure 6 depicts an IG G with six rules that each map to an IHC whose first external node is `null` and whose second external node is the root of a tree-like graph. Indices of edges not labeled with B are omitted for readability.

The sets of graphs modeled by IGs are defined similarly to languages of context-free word grammars (CFG) in which a nonterminal is replaced by a finite string: An IG derivation replaces an edge, say e, that is labeled with a nonterminal by a finite graph, say K. However, since arbitrarily many nodes may be attached to edge e, we have to clarify how the original graph and K are glued together. Hence, we identify each node attached to edge e with an external node of K (according to their position in both sequences). Formally,

[3] External nodes are needed to define the semantics of nonterminal edges.

[4] I.e., $v_{\texttt{null}} = ext(1)$ and for each $e \in E$ with $lab(e) \in N$, we have $att(e)(1) = v_{\texttt{null}}$.

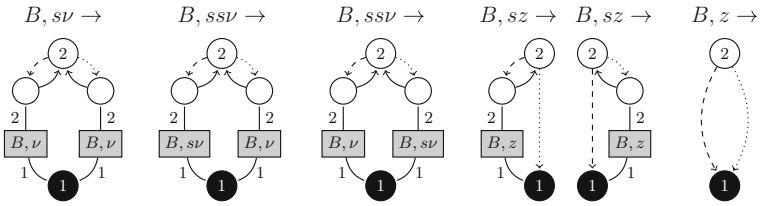


Fig. 6. An indexed hyperedge replacement grammar for balanced binary trees

Definition 3. *Let H, K be IHCs with pairwise disjoint sets of nodes and edges. Moreover, let $e \in E_H$ be an edge with $rank(lab_{E_H}(e)) = |ext_K|$. Then the* replacement *of e in H by K is given by $H\,[e \mapsto K] = (V, E, att, lab, ind, ext)$, where*

$$V = V_H \cup (V_K \setminus ext_K) \qquad E = \underbrace{(E_H \setminus \{e\})}_{=E'} \cup E_K$$

$$lab = (lab_H \restriction E') \cup lab_K \qquad ind = (ind_H \restriction E') \cup ind_K$$

$$att = (att_H \restriction E') \cup (att_K \fatsemi mod) \qquad ext = ext_H$$

where mod replaces each external node by the corresponding node attached to e.[5]

The above is the standard definition of hyperedge replacement in which indices and edge labels are treated the same (cf. [14]). It is then tempting to define that an IG G derives K from H if and only if there exists an edge $e \in E_H$ and a rule $(lab_H(e), ind_H(e) \to R) \in G$ such that K is isomorphic to $H\,[e \mapsto R]$. However, this notion is too weak to model balanced trees. In particular, since an index is treated as just another label, we cannot apply a derivation if the index of an edge does not exactly match an index on the left-hand side of an IG rule.

Instead, we use a finite prefix of indices in derivations and hide the remainder in variable ν. For example, assume an IG contains a rule $B, ss\nu \to R$. Given an edge with label B and index $\sigma = sssz$, an IG derivation may then hide sz in ν. The resulting index is $ss\nu$ and a derivation as defined naively above is possible. Finally, all occurrences of ν are replaced by the hidden suffix sz again.

To formalize indexed derivations, two auxiliary definitions are needed: Given a set $M \subseteq \mathbf{Types}$, we write E_H^M to refer to all edges of H that are labeled with a symbol in M, i.e. $E_H^M = \{e \in E_H \mid lab_H(e) \in M\}$. We write $H\,[\nu \mapsto \rho]$ to replace all occurrences of ν in (the index function ind of) H by ρ.[6]

Definition 4. *Let G be an IG and H, K be IHCs. Then G* directly derives *K from H, written $H \Rightarrow_G K$, if and only if either*

- *there exists a rule $(X, \sigma \to R) \in G$ and an edge $e \in E_H^{\{X\}}$ such that $ind_H(e) = \sigma$ and K is isomorphic to $H\,[e \mapsto R]$, or*

[5] $f \restriction M$ denotes the restriction of function f to domain M and $(f \fatsemi g)(s) = g(f(s))$. Moreover, function $mod = \{ext_K(k) \mapsto att_H(e)(k) \mid 1 \le k \le |ext_K|\} \cup \{v \mapsto v \mid v \in V \setminus ext_K\}$ is lifted to sequences of nodes by pointwise application.

[6] $H\,[\nu \mapsto \rho] = (V_H, E_H, att_H, lab_H, ind, ext_H)$ with $ind = \{ind_H(e)\,[\nu \mapsto \rho] \mid e \in E_H\}$.

– *there exists a rule* $(X, \sigma\nu \to R) \in G$*, an edge* $e \in E_H^{\{X\}}$*, and a sequence* $\rho \in I^+$ *such that* $ind_H(e) = \sigma\rho$ *and* K *is isomorphic to* $H\,[e \mapsto R\,[\nu \mapsto \rho]]$*.*

The reflexive, transitive closure of $\Rightarrow_G$ *is denoted by* $\Rightarrow_G^*$*. The inverse of* $\Rightarrow_G$ *is given by* ${}_G\Leftarrow$*. Finally,* $H\,{}_G\not\Leftarrow$ *iff there exists no* K *such that* $H\,{}_G\Leftarrow K$*.*

The *language* of an IG and an IHC H is the set of all graphs that can be derived from H and that do not contain nonterminals. Conversely, the inverse language of H is obtained by exhaustively applying inverse derivations to H.

Definition 5. *The* language L_G *and the* inverse language L_G^{-1} *of IG* G *are given by the following functions mapping indexed graphs to sets of indexed graphs:*

$$\begin{aligned} L_G(H) &= \{K \mid H \Rightarrow_G^* K \text{ and } E_K^N = \emptyset\}, \text{ and} \\ L_G^{-1}(H) &= \{K \mid H\,{}_G\Leftarrow^* K \text{ and } K\,{}_G\not\Leftarrow\}. \end{aligned}$$

For instance, the language of the IG in Fig. 6 for an IHC consisting of one edge labeled with B, ssz is the set of all balanced binary trees of height two.

To ensure existence of inverse languages and thus termination of abstraction, we assume that all rules of an IG G are *increasing*, i.e. for each rule $(X, \sigma \to H) \in G$ it holds that $|V_H| + |E_H| > rank(X) + 1$. As an example, notice that all rules of the IG in Fig. 6 are increasing. This amounts to a syntactic check on all rules that is easily discharged automatically. We conclude our introduction of IGs with a collection of useful properties.

Theorem 1. *Let* G *be an IG and* H *be an IHC over* N *and* I*. Then:*

1. $H \Rightarrow_G^* K$ *implies* $L_G(K) \subseteq L_G(H)$*.*
2. $L_G(H) = \begin{cases} \{H\} & \text{if } E_H^N = \emptyset \\ \bigcup_{H \Rightarrow_G K} L_G(K) & \text{otherwise.} \end{cases}$
3. *It is decidable whether* $L_G(H) = \emptyset$ *holds.*
4. *The inverse language* $L_G^{-1}(H)$ *of an increasing IG* G *is non-empty and finite.*

The first two properties are crucial for proving our analysis sound. The remaining properties ensure that we can construct well-defined (inverse) languages.

4 Abstract Domain

Our analysis is a typical forward abstract interpretation [12] that is parameterized by a user-supplied IG G. Its concrete domain consists of all IHCs without nonterminals. The abstract domain contains all IHCs to which no inverse IG derivation is applicable. The order of our abstract domain is language inclusion. Concretization γ and abstraction α correspond to computing the language and the inverse language of G, respectively. Our setting is summarized in Fig. 7.

$$\text{Concrete domain: } (\mathbf{Con} = \mathcal{P}(\underbrace{L_G(\mathbf{IHC})}_{\text{concrete IHCs}}), \subseteq) \qquad \text{Concretization: } \gamma = L_G$$

$$\text{Abstract domain: } (\mathbf{Abs} = \mathcal{P}(\underbrace{L_G^{-1}(\mathbf{IHC})}_{\text{fully abstract IHCs}}), \sqsubseteq), \quad \text{Abstraction: } \alpha = L_G^{-1}$$

Fig. 7. Concretization, abstraction and the respective domains for a given IG G. Here, $\sqsubseteq$ is given by $H \sqsubseteq K$ iff $\gamma(H) \subseteq \gamma(K)$. $\mathcal{P}(M)$ is the powerset of M. This setting yields a Galois connection for backward confluent IGs (cf. Sect. 5).

The concrete semantics of common imperative programs amounts to straightforward graph transformations. Let us assume that `Progs` is the set of all programs. Moreover, assume the concrete semantics of each program $P \in$ `Progs` is given by a (partial) function $\mathcal{C}[\![P]\!] : \mathbf{IHC} \to \mathbf{IHC}$ that captures the effect of executing P on an IHC. For example, step (2) in Sect. 2 computes $\mathcal{C}[\![\texttt{n = n.right}]\!]$.

As is standard, our analysis performs a fixed-point iteration of the abstract semantics that overapproximates the concrete semantics. Following the terminology of [20], our abstract semantics consists of three phases: materialization, execution of the concrete semantics, and canonicalization. That is, our abstract semantics is a function of the form $\mathcal{A}[\![.]\!] : \texttt{Progs} \to \mathbf{Abs} \to \mathbf{Abs}$ that is defined inductively on the structure of programs. In particular, for an atomic program $P \in$ `Progs`, we have $\mathcal{A}[\![P]\!] = \mathit{materialize}[\![P]\!] \fatsemi \mathcal{C}[\![P]\!] \fatsemi \mathit{canonicalize}[\![P]\!]$.[7]

Although materialization and canonicalization naturally depend on the user-provided grammar G, for readability we tacitly omit adding G as a parameter. Materialization ensures applicability of the concrete semantics by partially concretizing an IHC. It is thus a function $\mathit{materialize}[\![.]\!] : \texttt{Progs} \to \mathbf{IHC} \to \mathcal{P}_{\mathit{finite}}(\mathbf{IHC})$ that, for a given program, maps an IHC to a finite set of IHCs. Intuitively, materialization applies derivations $\Rightarrow_G$ until the concrete semantics can be applied (cf. Theorem 1.2). A detailed discussion of suitable materializations that are derived from a grammar G is found in [15,17]. In this paper, we consider a sufficient condition to ensure soundness.

Definition 6. *For every atomic program $P \in$ `Progs`, we require a materialization function $\mathit{materialize}[\![.]\!]$ such that $\gamma \fatsemi \mathcal{C}[\![P]\!] \dot{\subseteq} \mathit{materialize}[\![P]\!] \fatsemi \mathcal{C}[\![P]\!] \fatsemi \gamma$.*

Here, $\dot{\subseteq}$ denotes pointwise application of $\subseteq$. Examples of applying materialization are provided in steps (1) and (3) of Sect. 2.

Conversely to materialization, canonicalization takes a partially concretized program state and computes an abstract program state again. It is thus a function of the form $\mathit{canonicalize}[\![.]\!] : \texttt{Progs} \to \mathbf{IHC} \to \mathbf{Abs}$.

Definition 7. *For every program $P \in$ `Progs`, we require a canonicalization function $\mathit{canonicalize}[\![.]\!]$ such that $\gamma \mathrel{\dot{\subseteq}} \mathit{canonicalize}[\![P]\!] \fatsemi \gamma$.*

By Theorem 1(1), inverse IG derivations as well as the abstraction function α are suitable candidates for canonicalization. Examples of applying canonicalization are provided in steps (5) and (6) of Sect. 2.

[7] $f \fatsemi g$ denotes sequential composition of f and g, i.e. $(f \fatsemi g)(s) = g(f(s))$.

Assuming suitable materialization and canonicalization functions as of Definitions 6 and 7, our abstract semantics $\mathcal{A}[\![.]\!]$ computes an overapproximation of the concrete semantics $\mathcal{C}[\![.]\!]$:

Theorem 2 (Soundness). *For all* $P \in \texttt{Progs}$, $\gamma \fatsemi \mathcal{C}[\![P]\!] \mathrel{\dot\subseteq} \mathcal{A}[\![P]\!] \fatsemi \gamma$.

The quality of our analysis depends, naturally, on the quality of the user-defined grammar. That is, the better our grammar matches the data structures employed by a program, the more precise the results obtained from our analysis. In particular, our analysis does not necessarily terminate. For example, we cannot analyze a program working on doubly-linked lists if the user-supplied IG models trees only. As usual, termination has to be ensured by some sort of widening. In the simplest case, termination is achieved by fixing a maximal size of IHCs a priori. Whenever an IHC exceeds the fixed size, the analysis stops.

5 Backward Confluent IGs

Two components of our analysis are particularly involved: First, the inverse language of an IHC with respect to an IG has to be computed repeatedly during canonicalization, i.e. we have to exhaustively apply inverse IG derivations. Applying inverse derivations in turn requires finding isomorphic subgraphs in an IHC that can be replaced by a hyperedge. Since the subgraph isomorphism problem is well-known to be NP-complete, canonicalization is expensive.

Second, computing a fixed point requires us to check for language inclusion. However, the language inclusion problem for IGs is undecidable as it is already undecidable for context-free string grammars [5]. Undecidability of inclusion is common in the area of shape analysis, where supported data structures are either severely restricted to obtain decidability, or approximations are used.

We now discuss a subclass of IGs that addresses both problems:

Definition 8. *An IG* G *is* backward confluent *iff for all IHCs* H *the inverse language* $L_G^{-1}(H)$ *is a singleton set, i.e.* $|L_G^{-1}(H)| = 1$.

The definition of backward confluent IGs is, admittedly, rather semantics-driven. In particular, it solves the problem of expensive canonicalizations directly: Since the inverse language of an IHC is unique it suffices to exhaustively apply inverse derivations instead of trying all possible combinations. Fortunately, as shown in [19], backward confluence can be checked automatically. In particular, we constructed backward confluent IGs for singly- and doubly-linked, (a)cyclic lists, (balanced) trees (w/o back pointers), in-trees, lists of lists, and (in-)trees with linked leaves. In general, however, the class of graph languages generated by backward confluent IGs is strictly smaller than the class of languages generated by arbitrary IGs.

We now turn to our second desired property: a decidable inclusion problem. This property relies on the observation that two IHCs H, K that cannot be abstracted further, i.e. $H, K \,{}_G\!\not\Leftarrow$, are either isomorphic or have disjoint languages.

Theorem 3. *Let G be a backward confluent IG. Moreover, let $H, K \in$ **IHC** such that $K \,{}_G\!\not\Leftarrow$. Then it is decidable whether $L_G(H) \subseteq L_G(K)$ holds.*

To conclude this section, we remark that, for backward-confluent IGs, our concrete and abstract domain (cf. Fig. 7) form a Galois connection, i.e. our analysis falls within the classical setting of abstract interpretation [11].

6 Global Index Abstraction

The goal of our shape analysis is to enable reasoning about complex data structures, such as balanced binary trees. However, we might encounter infinitely many IHCs that vary in their indices only, thus preventing termination (cf. steps (1) and step (6) in Sect. 2). Our abstraction is thus often too precise.

To capture that an IHC models balanced trees, however, it suffices to keep track of the *differences* between indices: Assume, for example, that a node has two subtrees specified by nonterminal edges with indices sz and ssz. If we replace these indices by ssz and $sssz$, the underlying trees remain balanced.

Hence, we propose an index abstraction on top of IG-based abstraction. Intuitively, this abstraction removes a common suffix from all indices and replaces it by a placeholder. Apart from balancedness, it is applicable to properties such as "all sublists in a list of lists have equal length". The abstraction is again formalized by grammars; right-linear context-free word grammars (CFG) to be precise. Thus, let $I = I_N \cup I_T$ be a finite set of index symbols that is partitioned into a set of nonterminals I_N and a set of terminals I_T including the end-of-index symbol z. We call an index $\sigma \in I^+$ *well-formed* if $\sigma \in (I_T \setminus \{z\})^*(I_N \cup \{z\})$. That is, a well-formed index always ends with a nonterminal or the end-of-index symbol z. Accordingly, an IHC is well-formed if all of its indices are. We assume all indices — including indices in CFG rules — to be well-formed. Hence, all considered CFGs are right-linear and thus generate regular languages. We do not allow nonterminal index symbols in IGs, i.e. we assume for each IG rule $X, \sigma \to H$ that $ind_H(E_H^N) \subseteq I_T^*\{z, \nu\}$, where ν has been introduced in Definition 2.

To maintain relationships between indices, such as their difference, we require that all indices ending with the same nonterminal of an IHC are modified simultaneously. This leads us to a notion of global derivations and global languages.

Definition 9. *Let $H, K \in$ **IHC**. A CFG C* globally derives *K from H, written $H \rightrightarrows_C K$, if and only if there exists a rule $(X \to \tau) \in C$ such that $ind_H(E_H^N) \subseteq I_T^* I_N$ and K is isomorphic to $H[X \mapsto \tau]$, i.e. H in which all occurrences of X are replaced by τ. Again, $\rightrightarrows_C^*$ the reflexive, transitive closure of $\rightrightarrows_C$. ${}_C\!\Leftarrow$ denotes inverse derivations and ${}_C\!\not\Leftarrow$ that no inverse derivation is possible.*

Definition 10. *The* global language *and the* inverse global language *of a right-linear CFG C over I are given by:*

$$GL_C : \mathbf{IHC} \to \mathcal{P}(\mathbf{IHC}),\ H \mapsto \{K \mid H \rightrightarrows_C^* K \text{ and } ind_K(E_K^N) \subseteq I_T^+\}$$
$$GL_C^{-1} : \mathbf{IHC} \to \mathcal{P}(\mathbf{IHC}),\ H \mapsto \{K \mid H \,{}_C\!\Leftarrow^* K \text{ and } K \,{}_C\!\not\Leftarrow\}$$

Global derivations enjoy the same properties as IG derivations (Theorem 1). These properties are crucial to ensure soundness and termination of abstraction.

To combine global derivations and IG derivations, we consider a new derivation relation of the form $(\Rightarrow_G \cup \rightrightarrows_C)^*$. We can further simply this relation, because global derivations and IG derivations enjoy an orthogonality property:

Theorem 4. *$H\,(\Rightarrow_G \cup \rightrightarrows_C)^*\,K$ if and only if $H\,(\rightrightarrows_C^* \fatsemi \Rightarrow_G^*)\,K$.*

Thus, for materialization, it suffices to first apply global derivations and then apply IG derivations. Conversely, for abstraction, it suffices to first apply inverse IG derivations and then apply inverse global derivations.

It is then straightforward to refine our analysis from Sect. 4 by using the above derivation relation. To conclude this section, we remark that all results from Sects. 4 and 5 can be lifted to the refined analysis.

7 Implementation

We implemented our analysis in Attestor [3] to analyze Java programs. The source code and our experiments are available online.[8]

Input. Attestor supports a fragment of Java that includes recursive procedure calls, but no arithmetic. Apart from programs and grammars, linear temporal logic (LTL) specifications over execution paths can be supplied. Atomic propositions include heap shapes and reachability of variables (cf. [18]).

Output. Attestor generates a transition system in which each state consists of a program location and an IHC representing the abstract program state, i.e., a set of reachable heaps. This state space can also be explored graphically. Collecting the IHCs of all states with the same program location then coincides with the result of the abstract semantics presented in Sect. 4. Moreover, the tool applies LTL model-checking to verify provided LTL specifications.

Experimental Results. We evaluated our implementation against common challenging algorithms on various data structures and multiple LTL specifications. The results are shown in Table 1. Experiments were performed on an Intel Core i7-5820K at 3.30 GHz with the Java virtual machine limited to 2 GB of RAM. Program inputs covered all instances of the respective data structure through nonterminal edges for each employed data structure. In particular, `list to AVLTree` traverses a singly-linked list while inserting each of its elements into an (initially empty) AVL tree including all rebalancing procedures. Our implementation successfully verifies that the result is a balanced binary tree and the list has been completely traversed. This demonstrates that our analysis is capable of precisely reasoning about combinations of multiple data structures.

[8] https://github.com/moves-rwth/attestor-examples/releases/tag/v0.3.5-SEFM2018. Also confer the extended version [4].

Table 1. An excerpt of our experimental results. Provided verification times are in seconds including model-checking. Verified properties include *memory safety*, correct heap *shape* (including balancedness), correct return values, every element has been accessed, and the input data structure coincides with the output data structure. *Properties* refers to the worst runtime for verified LTL specifications.

Program	Mem. safety	Shape	Program	Mem. safety	Properties
AVL trees with parent pointers			Data structure traversals/other algorithms		
`binary search`	0.089	0.153	`List of cyclic lists`	0.115	0.115
`min. value`	0.128	0.204	`Tree (Lindstrom)`	0.084	11.60
`search and back`	0.140	0.158	`Skip list`	0.117	0.117
`search and swap`	0.823	1.106	`Tree (recursive)`	0.080	8.700
`rebalance`	1.500	1.769	`Zip list (recursive)`	0.118	0.118
`insert`	1.562	3.079	`DLL reversal`	0.054	0.126
`list to AVLTree`	1.784	1.892	`DLL insertion sort`	0.369	1.134

8 Related Work

Graph Transformations. Our work is an extension of an existing analysis based on context-free graph grammars [15]: From a theoretical perspective, IGs allow covering infinitely many context-free rules by a single nonterminal with an index variable. Covering infinitely many rules is essential when reasoning about relational properties, e.g. balancedness. From a practical perspective, our analysis is a standard forward abstract interpretation in contrast to previous approaches.

Separation Logic. The class of graphs described by context-free graph grammars is equivalent to a fragment of symbolic heap separation logic (SL) [16]. In contrast to SL, graph grammars give us access to a rich set of theoretical results from string and graph rewriting. For example, the concept of IGs is derived from Aho's indexed string grammars [2]. Moreover, the notion of backward confluence is well-studied in the context of graph rewriting (cf. [19]) and provides us with a decidable criterion to discharge entailments (language inclusion). Hip/Sleek uses SL enriched with arithmetic to specify size constraints on data structures (cf. [10]). Their focus is on program verification with user-supplied invariants. In contrast, our approach synthesizes invariants automatically. Furthermore, we provide decidable criteria for good data structure specifications whereas Hip/Sleek relies on heuristics to discharge entailments.

Static Analysis. [7,9] introduce a generic framework for relational inductive shape analysis based on user-supplied invariants. Applicability to red-black trees

is demonstrated in an example, but not covered by experiments. In [1], forest automata are extended by constraints between data elements associated with nodes of the heaps. The authors conjecture that their method generalizes to handle lengths of branches in a tree, which are needed to express balancedness properties. The details, however, are not worked out.

9 Conclusion

We developed a shape analysis that is capable of proving certain relational properties of data structures, such as balancedness of AVL trees. Our analysis is parameterized by user-supplied indexed graph grammars — a novel extension of context-free graph grammars. We implemented our approach and successfully applied it to common algorithms on AVL trees, lists, and combinations thereof.

References

1. Abdulla, P.A., Holík, L., Jonsson, B., Lengál, O., Trinh, C.Q., Vojnar, T.: Verification of heap manipulating programs with ordered data by extended forest automata. Acta Inf. **53**(4), 357–385 (2016)
2. Aho, A.V.: Indexed grammars - an extension of context-free grammars. J. ACM **15**(4), 647–671 (1968)
3. Arndt, H., Jansen, C., Katoen, J.P., Matheja, C., Noll, T.: Let this graph be your witness! an attestor for verifying Java pointer programs. In: CAV (2018, to appear)
4. Arndt, H., Jansen, C., Matheja, C., Noll, T.: Heap abstraction beyond context-freeness. CoRR abs/1705.03754 (2017). http://arxiv.org/abs/1705.03754
5. Bar-Hillel, Y., Perles, M., Shamir, E.: On formal properties of simple phrase structure grammars. Sprachtypologie und Universalienforschung **14**, 143–172 (1961)
6. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26:1–26:66 (2011)
7. Chang, B.E., Rival, X.: Relational inductive shape analysis. In: POPL 2008, pp. 247–260. ACM (2008)
8. Chang, B.E., Rival, X.: Modular construction of shape-numeric analyzers. EPTCS **129**, 161–185 (2013)
9. Chang, B.-Y.E., Rival, X., Necula, G.C.: Shape analysis with structural invariant checkers. In: Nielson, H.R., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 384–401. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-2_24
10. Chin, W., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. **77**(9), 1006–1036 (2012)
11. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977, pp. 238–252. ACM (1977)
12. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Log. Comput. **2**(4), 511–547 (1992)
13. Ferrara, P., Fuchs, R., Juhasz, U.: TVAL+ : TVLA and value analyses together. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 63–77. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_5

14. Habel, A.: Hyperedge Replacement: Grammars and Languages. LNCS, vol. 643. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0013875
15. Heinen, J., Jansen, C., Katoen, J., Noll, T.: Juggrnaut: using graph grammars for abstracting unbounded heap structures. Form. Method. Syst. Des. **47**(2), 159–203 (2015)
16. Jansen, C., Göbe, F., Noll, T.: Generating Inductive predicates for symbolic execution of pointer-manipulating programs. In: Giese, H., König, B. (eds.) ICGT 2014. LNCS, vol. 8571, pp. 65–80. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09108-2_5
17. Jansen, C., Heinen, J., Katoen, J.-P., Noll, T.: A local Greibach normal form for hyperedge replacement grammars. In: Dediu, A.-H., Inenaga, S., Martín-Vide, C. (eds.) LATA 2011. LNCS, vol. 6638, pp. 323–335. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21254-3_25
18. Jansen, C., Katelaan, J., Matheja, C., Noll, T., Zuleger, F.: Unified reasoning about robustness properties of symbolic-heap separation logic. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 611–638. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_23
19. Plump, D.: Checking graph-transformation systems for confluence. In: ECEASST, vol. 26 (2010)
20. Reps, T.W., Sagiv, M., Wilhelm, R.: Shape analysis and applications. In: Srikant, Y.N., Shankar, P. (eds.) The Compiler Design Handbook, 2nd edn. CRC Press, Boca Raton (2007)
21. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL 1999, pp. 105–118. ACM (1999)

Facilitating Component Reusability in Embedded Systems with GPUs

Gabriel Campeanu(✉)

Mälardalen Real-Time Research Center, Mälardalen University, Västerås, Sweden
gabriel.campeanu@mdh.se

Abstract. One way to fulfill the increased requirements (e.g., performance) of modern embedded systems is through the usage of GPUs. The existing embedded platforms that contain GPUs bring several challenges when developing applications using the component-based development methodology. With no specific GPU support, the component developer needs to encapsulate inside the component, all the information related to the GPU, including the settings regarding the GPU resources (e.g., number of used GPU threads). This way of developing components with GPU capability makes them specific to particular contexts, which negatively impacts the reusability aspect. For example, a component that is constructed to filter 640×480 pixel frames may produce erroneous results when reused in a context that deals with higher resolution frames. We propose a solution that facilitates the reusability of components with GPU capabilities. The solution automatically constructs several (functional-) equivalent component instances that are all-together used to process the same data. The solution is implemented as a state-of-the-practice component model (i.e., Rubus) and the evaluation of the realized extension is done through the vision system of an existing underwater robot.

1 Introduction

Modern embedded systems deal with huge amount of information that usually originates from the interaction with the environment. For example, the Google autonomous car[1] receives from its various sensors (e.g., camera, LIDAR, radar, ultrasound) an amount of 750 MB of data per second. This data is processed with enough performance in order for the car to be coordinated with the environment changes, such as moving pedestrians.

The embedded boards with Graphics Processing Units (GPUs) are feasible solutions for tackling the stringent requirements of modern embedded systems. Through its thousands of computational threads, the GPU is more efficient than the CPU when dealing with parallel data processing. For instance, a stereo matching application developed for embedded systems, delivers an increased frame rate processing when is executed on the GPU [9].

[1] https://waymo.com.

E. B. Johnsen and I. Schaefer (Eds.): SEFM 2018, LNCS 10886, pp. 287–301, 2018.
https://doi.org/10.1007/978-3-319-92970-5_18

Another trend in the embedded systems domain is the usage of the component-based development (CBD) for construction of systems [5]. This software engineering methodology promotes the development of systems through the composition of already existing software units called software components. CBD is successfully adopted by industry through various component models such as AUTOSAR [13], Rubus [8] and IEC 611-31 [10].

The existing component models for embedded systems development offer no specific GPU support. One way to develop components with GPU capabilities, is to encapsulate inside the components, all the GPU-related information. This leads to constructing components that are specific to particular contexts. A challenge appears when (re-)using the same component (that has particular GPU specifications encapsulated inside) in different applications. For instance, assuming we have a component that converts color frames into the grayscale format and has encapsulated a number of 640×480 GPU threads to use (i.e., one thread per pixel). When this component is (re-)used in applications that deal with 1024×960 pixel color images, it may result in providing erroneous results.

In this work, we provide a solution to increase the reusability of components with GPU capabilities, by constructing multiple instances of the same component in order to handle data of any size specification. For example, our solution generates three more instances of a component that filters 640×480 pixel frames in order to handle images with 2560×1920 pixels. The solution divides, via specific artifacts, an initial input data into several parts that are independently handled by the generated component instances. Based on the application design, the required artifacts and component instances are automatically generated into code during the system generation stage. We implement our solution as an extension of an existing industrial component model (i.e., Rubus) and evaluate it using an existing underwater robot case study.

The remainder of the paper is divided as follows. The background information covering GPUs and embedded system is presented in Sect. 2. The problem description (Sect. 3) and solution overview (Sect. 4) are presented using a running-case example. Section 5 describes the solution realization, while its evaluation is enclosed in Sect. 6. After we describe the related work (Sect. 7), we conclude the paper with a discussion section.

2 GPUs in Embedded Systems

The two main environments to develop GPU applications are CUDA[2] and OpenCL[3]. While CUDA is developed by NVIDIA to specifically address their hardware (i.e., NVIDIA GPUs), OpenCL is a general framework that addresses different processing units (e.g., mCPUs, GPUs, FPGAs) produced by various vendors. In this work, due to the fact that most of the existing embedded platforms with GPUs (developed by e.g., Intel, AMD, NVIDIA, Altera, IBM,

[2] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
[3] https://www.khronos.org/opencl/.

Samsung, Xilinx) support OpenCL, we utilize this particular environment for development of GPU functionality.

When using OpenCL to create GPU functionality, the developer needs to construct several hierarchical steps, as follows. At the highest level is the platform that contains the drivers. The platform addresses the existing devices, such as the GPU and the CPU. One of the device should be selected to execute the functionality. Memory buffers should be allocated in order to hold the input and/or output data of the functionality. The functionality, also know as the *kernel* may be constructed at any time, regardless of the required hierarchical steps. The arguments of the kernel are defined using the allocated input and output buffers. The number of threads used to execute the functionality are specified in such a way to match the platform characteristics, i.e., to not exceed the available thread resources. Furthermore, the threads are organized in particular groups (e.g., tiles of 8×8 threads) that directly influence the performance of the kernel execution.

Nowadays, the GPU is successfully integrated on various embedded platforms. Various vendors such as Intel, NVIDIA, Xilinx and AMD provide different embedded solutions with different characteristics, suitable for different domains. For example, while the NVIDIA Condor GR2[4] is utilized in high-performance solutions due to its high resources, the Mali-470 GPU with a low computation and energy consumption is employed in the construction of smart watches.

Component-based development is a software engineering methodology that promotes the development of applications through the composition of existing software units called components [5]. A core principle of CBD is the (re-)use of components in different contexts, which enhances the development efficiency. The communication between components is done through *interfaces*, which are specifications of the components' access points. In our work, we use *port-base* interfaces for sending/receiving data of different types.

An important aspect of CBD is the *encapsulation* fundamental, where all the component data and operations are encapsulated inside and hidden from anything outside. The only way to access the encapsulated information is through the component access points referred as *interfaces*. The way a component is constructed (alongside with its interfaces) is specified by a *component model*. Moreover, the component model defines the manner in which components communicate with each other, when composed into a system [4].

CBD is successfully adopted in industry through various component models. For real-time and embedded systems, the domain which we focus in this work, component models such as AUTOSAR [13], Rubus [8] and IEC 611-31 [10] are currently used in the development of applications. Particular interaction styles are employed by these component models when used for particular type of applications [6]. For example, the *pipe-and-filter* style, which is considered in this work, is suitable for streaming-of-events type of applications and adopted by e.g., Rubus and IEC 611-31 component models [6].

[4] http://www.eizorugged.com/products/vpx/condor-gr2-3u-vpx-rugged-graphics-nvidia-cuda-gpgpu/.

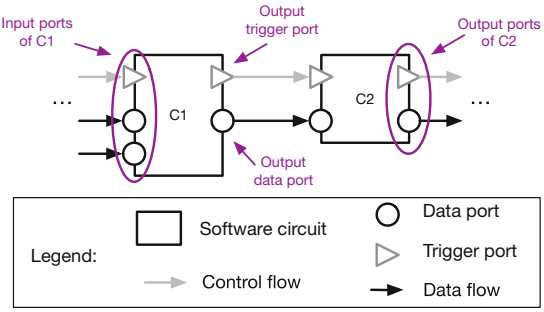


Fig. 1. Two connected Rubus components

As the Rubus component model is used in this work to realize our solution and for evaluation purposes, we introduce more details regarding it. The name of a Rubus component is *software circuit*. Each component is equipped with one input and output trigger port and one or several input and output data ports. Through these distinct types of ports (i.e., control and data ports), the component model provides a separation between control and data flow of the system, which allows an easy mapping between the control specifications of real-time and embedded systems, and the interaction model. Figure 1 illustrates two connected Rubus components characterized by input and output (trigger and data) ports. A Rubus component follows the Read-Execute-Write execution model. For example, *C1* is initially in an idle state. After receives the control of execution through its input trigger port, it Reads the input information via its two input data ports, Executes its functionality and Writes the result in the out data port. Finally, it passes the execution control to *C2* through the output trigger port and re-enters in the idle mode.

3 Problem Description

The existing component models used in the development of embedded systems provide no specific GPU support. Accordingly, the component developer, when constructing components with GPU functionality, needs to encapsulate all the GPU-related information inside the components. In this work, we focus on the encapsulated GPU settings regarding the GPU computation resources. For instance, for a component that filters frames of 640×480 pixels, the developer needs to hard-code inside the component a number of 640 * 480 GPU threads, where each thread processes a pixel. Moreover, the specified threads need to be grouped in a particular way (e.g., tiles of 8 by 8 threads) in order to match the size of the processed data. The grouping settings have a direct impact over the system performance.

A challenge comes when a component that contains hard-coded GPU settings is (re-)used in different contexts. Due to its encapsulated settings, the component

may produce erroneous results when dealing with data of different characteristics. For example, assuming there is a component that filters images and has encapsulated settings corresponding to 640×480 pixel frames. When the component is (re-)used in an application that handles 1024×1024 pixel images, it would be able to process only a part of the system data. An alternative would be to construct a component encapsulating the same functionality, but different GPU settings corresponding to 1024×1024 pixel images. Constructing components that can be used only in certain contexts, would significantly decrease the benefits of adopting CBD for construction of embedded systems with GPUs.

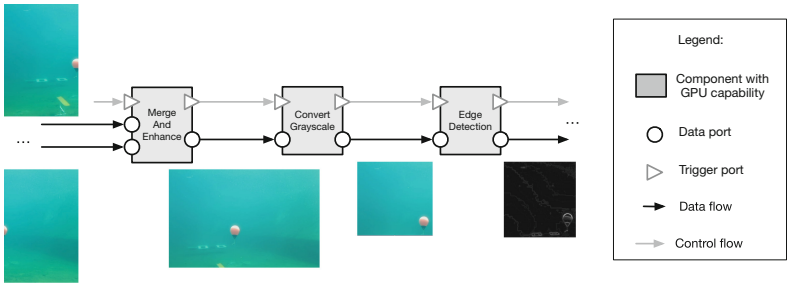


Fig. 2. Fragment of the component-based vision system of an underwater robot

To exemplify the challenge discussed in this work, we introduce a running-case example. We use a part of the vision system of an underwater robot, as described by Fig. 2. The vision system is constructed using the Rubus component model. The underwater robot is equipped with two cameras which provide a continuous flow of frames. Each pair of frames is received by the *MergeAndEnhance* component that merges and reduces the frames' noise. The resulted merged frame is converted to a grayscale format by *ConvertGrayscale* component and forwarded to *EdgeDetection* component that outputs a black-and-white frame, where objects are delimited by while lines.

We assume that the components were reused from different applications, and they were initially constructed with different GPU settings, as follows:

- *MergeAndEnhance* processes two 300×400 pixel frames and produces one 600×400 pixel frame,
- *ConvertGrayscale* converts 300×300 pixel frames, and
- *EdgeDetection* filters 600×500 pixel frames.

In the current example, the physical cameras provide frames with 300×450 pixels. The flow of the frames through the system including the (erroneous) outputs are illustrated in Fig. 2. We notice that the output of *EdgeDetection* is actually a portion of the (merged) frame produced by the *MergeAndEnhance* component.

4 Solution Overview

In order to facilitate reusability of components with GPU functionality in different contexts, we provide the following solution. Based on the design information, a component is instantiated with a number of identical instances in order to process data of various sizes. Each instance independently handles a part of the original data. The output processed data from all the instances are automatically merged together into a single one that the component communicates to its connected components.

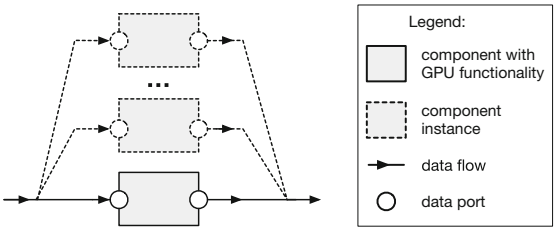


Fig. 3. Overview of the proposed solution

The overview of the proposed solution is illustrated in Fig. 3, where a component is duplicated into several more instances (illustrated in dashed lines) in order to handle income data with characteristics that are not supported by the component. The proposed solution automatically:

- decides on how many component instances are required,
- distributes data to each instance, and
- gathers the outcome of each instance into a single output data.

The instances are automatically generated, in a transparent way.

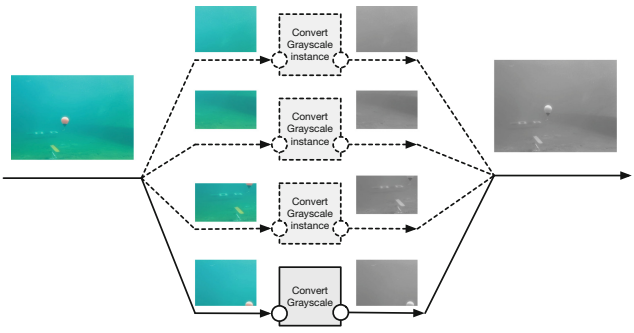


Fig. 4. Instances of the ConvertGrayscale component

Using the vision system running case, we describe our solution applied on the *ConvertGrayscale* component, as illustrated in Fig. 4. The solution automatically constructs three more component instances in order to correctly process the input received frame. The output grayscale frames provided by the used instances are merged together into a single frame which represents the conversion output of the *ConvertGrayscale* component.

5 Realization

In this work, we use white-box components, i.e., components with readable source code such as Rubus and IEC 61131 components. Furthermore, we target component models that follow the *pipe-and-filter* architecture style due to the embedded systems targeted by our solution (e.g., real-time, control-type applications). In this context, we provide a solution to automatically facilitate the (re-)use of components with GPU capability in different contexts. The solution is implemented as an extension of a state-of-the-practice component model (i.e., Rubus). Moreover, we use the OpenCL environment to implement components with GPU capability.

For each component with GPU capability, our solution checks the application design, i.e., the characteristics of the input ports and their received data. When a mismatch appears, the solution computes the number of required instances to handle the received data, and realize them inside the component, using the existing Rubus generation rules.

In order to divide the data, we introduce an artifact called *fork*. When a component is instantiated (i.e., differences between the income data attributes and component capabilities) a *fork* element is created. In order to not introduce additional component model elements, we use the existing Rubus framework and realize the *fork* artifact as a regular component equipped with input and output (trigger and data) ports. Based on the number of the input data ports of a component with GPU capability, the connected artifact will be equipped with an appropriate number of (input and output) data ports to handle all the data connections.

Similarly, the system generates an artifact called *join* to gather the outcomes from all component instances into one single outcome. Based on the number of output data ports of the component with GPU capabilities, the *join* artifact will be equipped with an appropriate number of ports to carry out the component communication. The *join* component is realized as a Rubus component, in an automatic and transparent manner. The *fork* and *join* are generated for each component with GPU capabilities, when needed.

Our solution handles the re-wiring between the introduced *join/fork* artifacts and the component and its created instances. The existing Rubus rules regarding component wiring are modified, and we introduce new rules that link the interfaces of the *join/fork* components with the generated interfaces of the component instances, and rest of the system. Moreover, in order to not introduce additional overhead for the system designer, the introduced components and their connections are transparently generated.

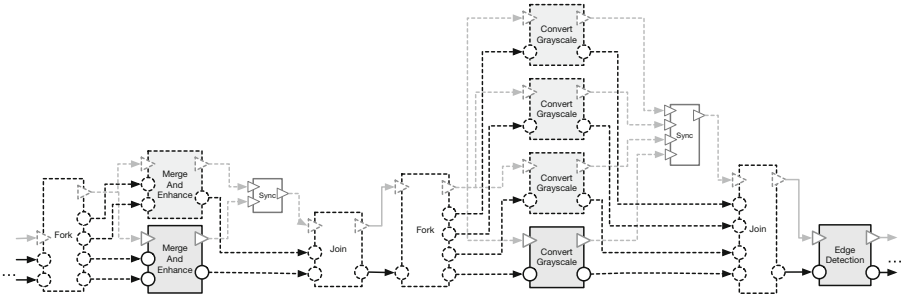


Fig. 5. The solution realization of the vision system

Figure 5 describes the solution realization of the vision system running-case. For the *MergeAndEnhance* component, a *fork* component is automatically generated and divides the input data into two parts which are provided to the *MergeAndEnhance* component and its instance. In order to connect to the two component instances[5], the *fork* component is equipped with:

- two input data ports that correspond to the number of *MergeAndEnhance* input data ports, and
- four output data ports through which it provide data to the two connected instances.

To gather all the outcomes, a *join* component is generated to copy the two results into one single location. The *join* component has:

- two input data ports, where each one receives the output data of the two generated component instances, and
- one output data port through which it sends the (gathered into one) result to the *ConvertGrayscale* component.

In a similar way, a *fork* component is created to divide and provide the correct data to the four *ConvertGrayscale* instances, and a *join* component to gather all the four results into a single frame. The system does not need to create any *fork*/*join* components for *EdgeDetection* due to its specifications, i.e., its functionality can handle the inputed data frame.

5.1 Implementation

The solution, based on the design of the system, constructs the proposed artifacts at the generation system stage, presented in the following paragraphs.

A Rubus software component is characterized by a header and several C source files that contain the definitions and declarations for:

[5] For simplification, we use the term of *instances* to refer to a component and its instance(s).

– an interface that contains structures used to define the input and output data ports,
– a constructor that initiate the resource requirements of the component,
– a behavior function that defines the component functionality, and
– a destructor that releases the allocated resources of the component.

The same Rubus rules that generate regular components (with GPU capability) are followed when implementing the component instances. For example, the implementation of the *MergeAndEnhance* component is identical with its instance.

In the next paragraphs, we describe the implementation of the *join* and *fork* components. The existing Rubus rules regarding generation of component interface are used to implement the interface of *fork* elements, as follows. Located in the header file, the interface contains a structure declaration that is composed of two (structure) elements corresponding to the input and output data ports. Figure 6 presents the interface of the fork component, i.e., *SWC_fork_MergeAndEnhance* (lines 34–37). The interface contains two elements:

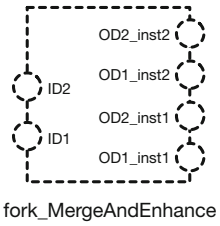
– *IP_SWC* - a structure with two elements that correspond to the input data ports, and
– *OP_SWC* - a structure with four elements corresponding to the output data ports of the fork component.

The two input ports receive the input data of *MergeAndEnhace* component, i.e., data locations and width and height dimensions of the input images. The output ports will contain, after the execution of the behavior function, the details of the data sent to each of the component instance. In a similar way, the interface of the *join_MergeAndEnhance* component will be automatically constructed.

In general, we let the system to generate the constructor for the *fork* component as it does not have distinctive requirements. On the other hand, the *join* component constructor needs special attention due to the artifact purpose, i.e., to copy several data into one location. Therefore, the *join* constructor needs to manage the allocation of a memory space to hold all of the combined data. This is realized with the OpenCL function *clCreateBuffer*[6] that allocates memory space on the GPU. The generation of the *join_MergeAndEnhance* constructor is described in Listing 1.1, where the width and height sizes of the outcome image are calculated using the corresponding values of the frames received through each input port (i.e., lines 3 and 8). The computed width and height sizes are used to allocate a memory space (i.e., line 11) that is big enough to contain all of the received data.

The behavior function of a *fork* component, based on the information of the input data and the number of connected component instances and their characteristics, computes the characteristics (i.e., location, width and height) of the data that will be processed by each component instance. For the

[6] https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clCreateBuffer.html.




```
typedef struct {
  GPU_unsigned_char *ptr;
}data_type;

typedef struct {
  int size;
}dim1;

typedef struct {
  int size;
}dim2;

typedef struct {
  data_type *place;
  dim1 width;
  dim2 height;
}img_format;

/* input ports */
typedef struct {
  img_format *ID1;
  img_format *ID2;
}IP_SWC;

/* output ports */
typedef struct {
  img_format OD1_inst1;
  img_format OD2_inst1;
  img_format OD1_inst2;
  img_format OD2_inst2;
}OP_SWC;

/* the interface declaration */
typedef struct {
  IP_SWC IP;
  OP_SWC *OP;
}SWC_fork_MergeAndEnhance;
```

Fig. 6. The interface of a *fork* component

fork_MergeAndEnhance component, the behavior function calculates two pointers that direct to two memory locations corresponding to two parts of the initial image. The output data ports are set with the computed pointers, alongside with the corresponding image dimensions (i.e., width and height).

Listing 1.1. The constructor of a *join* component

```
int width = 0;
<foreach input port p>
  int width += args->IP.p->width.size;
<endforeach>

int height = 0;
<foreach input port p>
  int height += args->IP.p->height.size;
<endforeach>

void *location = clCreateBuffer(contex, CL_MEM_WRITE_ONLY, 3*width*height, NULL, NULL);
```

The behavior function of the *join* component copies all the income data into a single memory location (allocated by the constructor). For our vision system example, the *join_MergeAndEnhance* behavior function copies two images using the OpenCL function *clEnqueueCopyBuffer*[7]. The specifications of the input

[7] https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clEnqueueCopyBuffer.html.

data (determined through the input port characteristics) and the memory location (allocated by the constructor) to hold the two images, are used as parameters for the copying activity. Listing 1.2 describes one of the copy activities done by the behavior function, i.e., the copy of the image received through the input port ID1. Similarly, another copy activity corresponding to the data received by the second input port, is generated inside the behavior function.

Listing 1.2. A part of the behavior function of a *join* component

```
clEnqueueReadBuffer(command_queue, args->IP.ID1->place->ptr, (unsigned char*) location, 0, 0, 3*(args->IP.ID
    ->width.size)*(args->IP.ID->height.size) * sizeof(unsigned char), 0, NULL, NULL);
```

Regarding the destructors, we do not include specific instructions for the *fork* destructor and let the system to handle it using the existing Rubus framework rules. For the *join* component, the destructor needs to release the memory allocated by the constructor. Therefore, for the generation of *join_MergeAndEnhance* destructor, we use the *clReleaseMemObject*[8] function to release the memory allocated by the constructor, as depicted in Listing 1.3.

Listing 1.3. The destructor of a *join* component

```
clReleaseMemObject(location);
```

6 Evaluation

In this section, we examine the feasibility of our solution using the introduced running case, i.e., the vision system of the underwater robot. Moreover, we analyze: *(i)* the execution overhead, and *(ii)* memory overhead introduced by the generation of the *fork* and *join* components, and the component instances.

We constructed two Rubus versions of the vision system, where one is constructed using our solution and the other, referred as the custom version, is constructed using regular components with hard-coded settings to explicitly handle the system input images. In all of the experiments, we make comparisons between the two versions. Furthermore, the GPU functionality of each component in both versions is constructed in such a way to handle also input frames that have lower (width and height) attributes than the component specifications. The platform used for the experiments in an embedded platform with an AMD Kabini SoC[9].

Listing 1.4 presents a part of the *ConvertGrayscale* functionality. For simplification purposes, we define three variables (i.e., lines 1–3) to hold the attributes of the input data (i.e., location, width and height). These variables are used as arguments for the *ConvertGrayscale_fct* function (referred as the GPU kernel) that contains the conversion-to-grayscale GPU functionality. The hard-coded settings, that correspond to the GPU thread index, are accessed from the kernel function using specific calls (i.e., *get_global_id*) in lines 10 and 11, and checked

[8] https://www.khronos.org/registry/OpenCL/sdk/1.1/docs/man/xhtml/clReleaseMemObject.html.

[9] https://unibap.com/product/advanced-hetereogeneous-computing-modules.

against the received kernel arguments (i.e., line 12). When the arguments values do not match (i.e., less than) the number of utilized threads, the extra threads are discarded (i.e., they do not execute).

Listing 1.4. Part of the *ConvertGrayscale* GPU functionality

```
1  unsigned char* in = args->IP.ID1->place->ptr;
2  int width = args->IP.ID->width.size;
3  int height = args->IP.ID->height.size;
4
5  __kernel void ConvertGrayscale_fct(__global const unsigned char *in, int width, int
       height, __global unsigned char *out)
6  {
7  /* compute absolute image position (x, y) */
8   int x =  get_global_id(0);
9   int y =  get_global_id(1);
10
11 /* relieve threads that are outside of the received image */
12  if (x >= width || y >= height) return;
```

The execution of the two vision system versions produced results (i.e., frames) that were identical. The introduced artifacts and activities (i.e., dividing and merging data) did not influence the system's outcomes.

In the second experiment, we focused on the execution time overhead of the introduced solution. Our solution introduces: *(i)* four artifacts (i.e., *fork* and *join* components), *(ii)* one additional *MergeAndEnhance* instance, and *(iii)* three more *ConvertGrayscale* instances. To examine the introduced overhead, we calculate the end-to-end execution of the two vision system versions. The execution time for the version implemented with our solution was 9.4 ms, while the custom version had 4.6 ms.

As the memory characteristic is a sensible topic in the embedded systems domain, in the last part of the evaluation we analyzed the memory overhead introduced by our solution. For the custom version of the vision system, the total memory requirement is of 494.2 kB, where *MergeAndEnhance* requires 177.6 kB of memory, *ConvertGrayscale* requires 175.6 kB of memory, and *EdgeDetection* requires 130.8 kB of memory. For the version that uses our solution, where there are two *MergeAndEnhance* instances, four *ConvertGrayscale* instances and one *EdgeDetection* instance, the total system requirement is of 722 kB of memory. We mention that a *MergeAndEnhance* instance that processes two 300×400 pixel frames requires 158.6 Kb of memory and a *ConvertGrayscale* instance that processes (at a time) one 300×300 pixel frame, requires 63.7 kB of memory. We notice that the custom version requires with 227 kB less memory than the version constructed with our solution.

Although the custom version has an improved execution time and requires less memory than the version that uses our solution, the components are specifically constructed for this application and platform, and have a low reusability in other (software and hardware) contexts.

7 Related Work

The increased requirements of modern applications lead to the adoption of heterogeneity in embedded systems. AUTOSAR component model, utilized in the

automotive industry, was extended with multi-core ECUs support [11]. Another solution to satisfy the increased demands of modern applications is to use accelerator hardware. The FPGA is one of the feasible solutions to be used as co-processor for data demanding applications. Andrews et al. proposes a way to use COTS components, referred as hybrid components, to develop applications for CPU-FPGA hardware [1].

The GPU in the context of embedded systems is addressed by Campeanu et al. which facilitate the development of applications for CPU-GPU hardware [3]. The authors proposed to enrich each component (with GPU capability) with a specific configuration interface. Through this interface, the component receives from e.g., the system designer, individual GPU settings regarding the number of GPU threads used by the functionality. We consider this as a possible solution to tackle the challenge discussed in this work, but it comes with two disadvantages, as follows. The system developer needs to have, at the time when designing the system, detailed information (i.e., the physical GPU threads limitation) of the hardware platform that will host the applications. The detailed hardware platform is not always known at design time. Another downside is that the system designer needs to: *(i)* have knowledge about GPU development, or *(ii)* correspond with the component developer (breaking the separation-of-concern CBD principle), in order to provide good/best GPU thread settings for the components with GPU capability. Although there is an overhead introduced by our solution (i.e., memory usage and execution time), we believe that our work increases the reusability aspect while preserving the separation-of-concern principle.

Some component models develop traditional (CPU-based) systems by hard-coding inside the components the specific characteristics of the hardware. Lednicki et al. introduce an additional layer (i.e., mapping layer) to address the flexibility of components [12]. The introduced mapping layer connects software and hardware platforms allowing them to be independently developed. We consider that new architectural elements (e.g., software layers) would greatly increase the overhead of Rubus solutions. Therefore, we constructed our solution using the existing elements of the Rubus framework.

It is worth to mention the work of Dastgeer et al. that introduce the PEPPHER framework that uses a component-based development approach to construct CPU-GPU systems [7]. They refer to a software component as a block that encapsulates one or several implementation variants. All the variants are computationally equivalent and have the same interface. Whenever the component is executed, a proper variant is selected based on the software call parameters and available hardware resources. In our work, we also use multiple instances of the same component; however, all of the instances are used in order to produce the correct output.

8 Discussion

The existing component models have no specific GPU support in development of embedded systems. This leads to constructing components, with hard-coded

settings, that are specific to certain contexts. To tackle this challenge, we propose a solution to improve the reusability of components with GPU capability, for different contexts. The solution introduces two types of artifacts, i.e. *fork* and *join* artifacts, that are automatically generated to divide and merge data. Moreover, the solution instantiates each component with GPU functionality with a number of instances in order to handle data of any size.

There are other solutions (e.g., see Sect. 7, second paragraph) to tackle the challenge discussed in this work. As presented in the evaluation, a component with GPU capability may be hard-coded with (high number of) GPU threads settings to handle images of different sizes. There are two limitations of this solution, as follows:

- each GPU platform has a limited number of threads (e.g., CL_DEVICE_MAX_WORK_GROUP_SIZE for OpenCL contexts). Hard-coding large thread resources inside components will make them specific to certain GPU platforms, with large available resources.
- Imposing large GPU thread usage for components with GPU capability used to process data of small-to-medium sizes, leads to waste of GPU resources.

Another downside of the proposed solution is the memory overhead. For each component that cannot handle the received data, we generate a number of instances, alongside the *join* and *fork* components. The vision system version implemented with our solution has generated four more component instances (i.e., one for *MergeAndEnhance* and three for *ConvertGrayscale*) and four *fork*/*join* components. We consider that the memory overhead is a major downside of our solution, due to the domain targeted of our work (i.e., real-time and embedded systems) where the system is characterized by limited resources (e.g., available memory). Therefore, one future work direction is to reduce the introduced memory overhead by grouping all the created instances in a conceptual component. Another future work direction is to increase the GPU parallelism level by simultaneous executing the instances of the same component. Using the existing approaches for parallel execution of components on GPU [2], we may improve the system performance, while delivering an increased component reusability.

Besides the overhead introduced by our solution (i.e., increased memory usage and execution time), we consider that our solution decreases the existing gap of component-based development for embedded systems with GPUs, facilitating the reusability of components with GPU capabilities.

Acknowledgment. The work is partially supported by the Knowledge Foundation through the ORION project (reference number 20140218).

References

1. Andrews, D., Niehaus, D., Ashenden, P.: Programming models for hybrid CPU/FPGA chips. Computer **37**(1), 118–120 (2004)
2. Campeanu, G.: Parallel execution optimization of GPU-aware components in embedded systems. In: The 29th International Conference on Software Engineering and Knowledge Engineering, SEKE 2017, 5–7 July 2017, Pittsburgh, USA (2017)
3. Campeanu, G., Carlson, J., Sentilles, S., Mubeen, S.: Extending the Rubus component model with GPU-aware components. In: 2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE). IEEE (2016)
4. Chaudron, M., Crnkovic, I.: Component-based software engineering. In: van Vliet, H. (ed.) Software Engineering: Principles and Practice. Wiley, Chichester (2008)
5. Crnkovic, I., Larsson, M.: Building Reliable Component-Based Software Systems. Artech House Inc., Norwood (2002)
6. Crnkovic, I., Sentilles, S., Vulgarakis, A., Chaudron, M.: A classification framework for software component models. IEEE Trans. Softw. Eng. **37**, 593–615 (2011)
7. Dastgeer, U., Li, L., Kessler, C.: The PEPPHER composition tool: performance-aware dynamic composition of applications for GPU-based systems. In: 2012 SC Companion High Performance Computing, Networking, Storage and Analysis (SCC). IEEE (2012)
8. Hanninen, K., Maki-Turja, J., Nolin, M., Lindberg, M., Lundback, J., Lundback, K.-L.: The Rubus component model for resource constrained real-time systems. In: International Symposium on Industrial Embedded Systems, SIES 2008. IEEE (2008)
9. Humenberger, M., Zinner, C., Weber, M., Kubinger, W., Vincze, M.: A fast stereo matching algorithm suitable for embedded real-time systems. Comput. Vis. Image Underst. **114**, 1180–1202 (2010)
10. John, K.-H., Tiegelkamp, M.: IEC 61131–3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-662-07847-1
11. Kluge, F., Gerdes, M., Ungerer, T.: AUTOSAR OS on a message-passing multicore processor. In: SIES, pp. 287–290 (2012)
12. Lednicki, L., Feljan, J., Carlson, J., Zagar, M.: Adding support for hardware devices to component models for embedded systems. In: The Sixth International Conference on Software Engineering Advances (2011)
13. AD Partnership: Technical overview v4.2. http://www.autosar.org. Accessed 14 Apr 2018

Author Index

Ábrahám, Erika 89
Arndt, Hannah 271

Babaee, Reza 205
Balasubramanian, Daniel 139
Barbon, Gianluca 173
Bendík, Jaroslav 189

Campeanu, Gabriel 287
Carlsson, Mats 239
Černá, Ivana 189
Chen, Tzu-Chun 73
Choi, Yunja 254
Cleaveland, Rance 37

Dagnat, Fabien 54
Din, Crystal Chang 73
Dubois, Catherine 239

Fischmeister, Sebastian 205

Ghassabani, Elaheh 189
Golra, Fahad Rafique 54
Grinchtein, Olga 239
Guerin, Sylvain 54
Gurfinkel, Arie 205

Hallerstede, Stefan 21
Harman, Mark 3
Hasan, Osman 223
Hasanagić, Miran 21

Jansen, Christina 271

Karsai, Gabor 139
Kersten, Rody 139
König, Jürgen 105
Kostyuchenko, Dmitriy 139
Krings, Sebastian 21

Larsen, Peter Gorm 21
Le Goues, Claire 155
Leroy, Vincent 173
Leuschel, Michael 21
Lindvall, Mikael 37
Luckow, Kasper 123, 139

Matheja, Christoph 271

Noll, Thomas 271

Păsăreanu, Corina S. 123
Pearson, Justin 239

Rashid, Adnan 223

Salaün, Gwen 173
Sayar, Imen 54
Schellhorn, Gerhard 105
Schlatte, Rudolf 73
Schulze, Christoph 37
Schupp, Stefan 89
Siddique, Umair 223
Souquières, Jeanine 54

Travkin, Oleg 105

van Tonder, Rijnard 155
Visser, Willem 123

Wedel, Monika 105
Wehrheim, Heike 105
Whalen, Michael 189