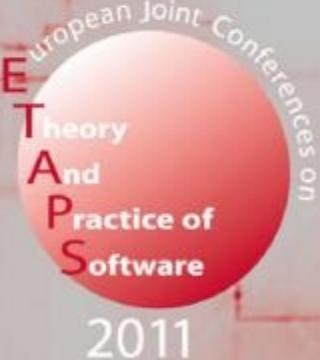


Dimitra Giannakopoulou
Fernando Orejas (Eds.)

Fundamental Approaches to Software Engineering

14th International Conference, FASE 2011
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2011
Saarbrücken, Germany, March/April 2011, Proceedings



Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison, UK

Takeo Kanade, USA

Josef Kittler, UK

Jon M. Kleinberg, USA

Alfred Kobsa, USA

Friedemann Mattern, Switzerland

John C. Mitchell, USA

Moni Naor, Israel

Oscar Nierstrasz, Switzerland

C. Pandu Rangan, India

Bernhard Steffen, Germany

Madhu Sudan, USA

Demetri Terzopoulos, USA

Doug Tygar, USA

Gerhard Weikum, Germany

Advanced Research in Computing and Software Science Subline of Lecture Notes in Computer Science

Subline Series Editors

Giorgio Ausiello, *University of Rome ‘La Sapienza’, Italy*

Vladimiro Sassone, *University of Southampton, UK*

Subline Advisory Board

Susanne Albers, *University of Freiburg, Germany*

Benjamin C. Pierce, *University of Pennsylvania, USA*

Bernhard Steffen, *University of Dortmund, Germany*

Madhu Sudan, *Microsoft Research, Cambridge, MA, USA*

Deng Xiaotie, *City University of Hong Kong*

Jeannette M. Wing, *Carnegie Mellon University, Pittsburgh, PA, USA*

Dimitra Giannakopoulou
Fernando Orejas (Eds.)

Fundamental Approaches to Software Engineering

14th International Conference, FASE 2011
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2011
Saarbrücken, Germany, March 26–April 3, 2011
Proceedings

Volume Editors

Dimitra Giannakopoulou
Carnegie Mellon University/NASA Ames Research Center
Moffett Field, CA 94035, USA
E-mail: dimitra.giannakopoulou@nasa.gov

Fernando Orejas
Universitat Politècnica de Catalunya
08034 Barcelona, Spain
E-mail: orejas@lsi.upc.edu

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-19810-6 e-ISBN 978-3-642-19811-3
DOI 10.1007/978-3-642-19811-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011922619

CR Subject Classification (1998): D.2.4, D.2, F.3, D.3, C.2, H.4, C.2.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

© Springer-Verlag Berlin Heidelberg 2011

Springer-Verlag Berlin Heidelberg 2001
This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

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

Foreword

ETAPS 2011 was the 14th instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised the usual five sister conferences (CC, ESOP, FASE, FOS-SACS, TACAS), 16 satellite workshops (ACCAT, BYTECODE, COCV, DICE, FESCA, GaLoP, GT-VMT, HAS, IWIGP, LDTA, PLACES, QAPL, ROCKS, SVARM, TERMGRAPH, and WGT), one associated event (TOSCA), and seven invited lectures (excluding those specific to the satellite events).

The five main conferences received 463 submissions this year (including 26 tool demonstration papers), 130 of which were accepted (2 tool demos), giving an overall acceptance rate of 28%. Congratulations therefore to all the authors who made it to the final programme! I hope that most of the other authors will still have found a way of participating in this exciting event, and that you will all continue submitting to ETAPS and contributing to make of it the best conference on software science and engineering.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis and improvement. The languages, methodologies and tools which support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a confederation in which each event retains its own identity, with a separate Programme Committee and proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronised parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for ‘unifying’ talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings.

ETAPS 2011 was organised by the *Universität des Saarlandes* in cooperation with:

- ▷ European Association for Theoretical Computer Science (EATCS)
- ▷ European Association for Programming Languages and Systems (EAPLS)
- ▷ European Association of Software Science and Technology (EASST)

It also had support from the following sponsors, which we gratefully thank: DFG DEUTSCHE FORSCHUNGSGEMEINSCHAFT; ABSINT ANGEWANDTE INFORMATIK GMBH; MICROSOFT RESEARCH; ROBERT BOSCH GMBH; IDS SCHEER AG / SOFTWARE AG; T-SYSTEMS ENTERPRISE SERVICES GMBH; IBM RESEARCH; GWSAAR GESELLSCHAFT FÜR WIRTSCHAFTSFÖRDERUNG SAAR MBH; SPRINGER-VERLAG GMBH; and ELSEVIER B.V.

The organising team comprised:

General Chair: *Reinhard Wilhelm*

Organising Committee: *Bernd Finkbeiner, Holger Hermanns* (chair),
Reinhard Wilhelm, Stefanie Haupert-Betz,
Christa Schäfer

Satellite Events: *Bernd Finkbeiner*

Website: *Hernán Baró Graf*

Overall planning for ETAPS conferences is the responsibility of its Steering Committee, whose current membership is:

Vladimiro Sassone (Southampton, Chair), Parosh Abdulla (Uppsala), Gilles Barthe (IMDEA-Software), Lars Birkedal (Copenhagen), Michael O’Boyle (Edinburgh), Giuseppe Castagna (CNRS Paris), Marsha Chechik (Toronto), Sophia Drossopoulou (Imperial College London), Bernd Finkbeiner (Saarbrücken) Cormac Flanagan (Santa Cruz), Dimitra Giannakopoulou (CMU/NASA Ames), Andrew D. Gordon (MSR Cambridge), Rajiv Gupta (UC Riverside), Chris Hankin (Imperial College London), Holger Hermanns (Saarbrücken), Mike Hinckey (Lero, the Irish Software Engineering Research Centre), Martin Hofmann (LMU Munich), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Jens Knoop (Vienna), Barbara König (Duisburg), Shriram Krishnamurthi (Brown), Juan de Lara (Madrid), Kim Larsen (Aalborg), Rustan Leino (MSR Redmond), Gerald Luettgen (Bamberg), Rupak Majumdar (Los Angeles), Tiziana Margaria (Potsdam), Ugo Montanari (Pisa), Luke Ong (Oxford), Fernando Orejas (Barcelona), Catuscia Palamidessi (INRIA Paris), George Papadopoulos (Cyprus), David Rosenblum (UCL), Don Sannella (Edinburgh), João Saraiva (Minho), Helmut Seidl (TU Munich), Tarmo Uustalu (Tallinn), and Andrea Zisman (London).

I would like to express my sincere gratitude to all of these people and organisations, the Programme Committee Chairs and members of the ETAPS conferences, the organisers of the satellite events, the speakers themselves, the many reviewers, all the participants, and Springer for agreeing to publish the ETAPS proceedings in the ARCoSS subline.

Finally, I would like to thank the Organising Chair of ETAPS 2011, Holger Hermanns and his Organising Committee, for arranging for us to have ETAPS in the most beautiful surroundings of Saarbrücken.

Preface

FASE (Fundamental Approaches to Software Engineering) is concerned with the foundations on which software engineering is built. Its focus is on novel techniques and the way in which they contribute to making software engineering a more mature and soundly based discipline. This year, we particularly encouraged contributions that combine the development of conceptual and methodological advances with their formal foundations and tool support. We welcomed contributions on all such fundamental approaches, including:

- Software engineering as an engineering discipline, including its interaction with and impact on society
- Requirements engineering: capture, consistency, and change management of software requirements
- Software architectures: description and analysis of the architecture of individual systems or classes of applications
- Specification, design, and implementation of particular classes of systems: adaptive, collaborative, embedded, distributed, mobile, pervasive, or service-oriented applications
- Software quality: validation and verification of software using theorem proving, model-checking, testing, analysis, refinement methods, metrics or visualization techniques
- Model-driven development and model-transformation: design and semantics of semi-formal visual languages, consistency and transformation of models
- Software processes: support for iterative, agile, and open source development
- Software evolution: re-factoring, reverse and re-engineering, configuration management and architectural change, or aspect-orientation

We solicited two types of contributions: research papers and tool demonstration papers. We received submissions from 31 countries around the world: 116 abstracts followed by 99 full papers, of which 2 were tool papers. The selection process was rigorous. Each paper received at least three reviews. We obtained external reviews for papers that lacked expertise within the Program Committee. We also had four reviews for all papers that did not receive high bids and for papers that had Program Committee authors so as to ensure high quality in accepted papers. Moreover, the Program Committee had extensive online discussions in order to decide on the papers to be accepted for the conference.

The Program Committee accepted 29 research papers, corresponding to a 29% acceptance rate among the full submissions. We believe that the accepted papers made a scientifically strong and exciting program, which triggered interesting discussions and exchange of ideas among the ETAPS participants. The accepted papers cover several aspects of software engineering, including modeling, specification, verification, testing, quality of service, code development, and model-based development.

Finally, FASE 2011 was honored to host an invited talk by Marta Kwiatkowska, titled “Automated Learning of Probabilistic Assumptions for Compositional Reasoning.” We feel that this talk will inspire the software engineering community towards two key trends in formal reasoning of realistic systems. Probabilistic reasoning is often the only meaningful approach in the presence of uncertainty, and compositionality is essential for scalability.

We would like to thank all authors who submitted their work to FASE. Without their excellent contributions we would not have managed to prepare a strong program. We also thank the Program Committee members and external reviewers for their high-quality reviews and their effort and time in making the selection process run smoothly and on time. Finally, we wish to express our gratitude to the Organizing and Steering Committees for their excellent support.

The logistics of our job as Program Chairs were facilitated by the EasyChair system.

January 2011

Dimitra Giannakopoulou
Fernando Orejas

Organization

Program Chairs

Dimitra Giannakopoulou
Fernando Orejas

Carnegie Mellon University/NASA Ames (USA)
Universitat Politècnica de Catalunya (Spain)

Program Committee

Josh Berdine	Microsoft Research Cambridge (UK)
Marsha Chechik	University of Toronto (Canada)
Shin-Chi Cheung	Hong Kong University of Science and Technology (China)
Juan De Lara	Universidad Autónoma de Madrid (Spain)
Claudia Ermel	Technische Universität Berlin (Germany)
José Luiz Fiadeiro	University of Leicester (UK)
Alex Groce	Oregon State University (USA)
Klaus Havelund	NASA / JPL (USA)
Reiko Heckel	University of Leicester (UK)
Mats Heimdahl	University of Minnesota (USA)
Paola Inverardi	Università dell'Aquila (Italy)
Valerie Issarny	INRIA Paris-Rocquencourt (France)
Joost-Pieter Katoen	RWTH Aachen University (Germany)
Jeff Magee	Imperial College London (UK)
Tom Maibaum	McMaster University (Canada)
Tiziana Margaria	Universität Potsdam (Germany)
Leonardo Mariani	University of Milano Bicocca (Italy)
Laurent Mounier	VERIMAG (France)
Corina Păsăreanu	Carnegie Mellon / NASA Ames (USA)
Gabriele Taentzer	Philipps-Universität Marburg (Germany)
Daniel Varró	Budapest University of Technology and Economics (Hungary)
Kapil Vaswani	Microsoft Research India (India)
Willem Visser	Stellenbosch University (South Africa)
Martin Wirsing	Ludwig-Maximilians-Universität München (Germany)
Andrea Zisman	City University London (UK)

External Reviewers

Marco Autili	Jonathan Heinen	Neha Rungta
Thorsten Arendt	Frank Hermann	Oliver Rüthing
Cyrille Valentin Artho	Ábel Hegedüs	Mehrdad Sabetzadeh
Mayur Bapodra	Ákos Horváth	Rick Salay
Howard Barringer	Stefan Jurack	Helen Schonenberg
Shoham Ben-David	Pierre Kelsen	Shalini Shamasunder
Enrico Biermann	Tamim Khan	Jocelyn Simmonds
Gábor Bergmann	Imre Kocsis	Élodie-Jane Sims
Laura Bocchi	Leen Lambers	Scott Smolka
Henrik Bohnenkamp	Yngve Lamo	Romina Spalazzese
Artur Boronat	Antónia Lopes	Matt Staats
Benjamin Braatz	Wendy MacCaull	Bernhard Steffen
Jacques Carette	Rodrigo Machado	Volker Stoltz
Robert Clarisó	Katharina Mehner	Mark Timmer
Roy Crole	Tony Modica	Massimo Tivoli
Davide Di Ruscio	Muhammad Naeem	Emilio Tuosto
Zinovy Diskin	Shiva Nejati	Tarmo Uustalu
Hartmut Ehrig	Thomas Noll	Frits Vaandrager
Ylies Falcone	Frank Ortmeier	Gergely Varró
Karsten Gabriel	Jun Pang	Arnaud Venet
Ulrike Golas	Patrizio Pelliccione	Xinming Wang
László Gönczy	Gergely Pinter	Gordon Wilfong
Andreas Griesmayer	Fawad Qayum	Chang Xu
Radu Grosu	István Ráth	Rongjie Yan
Lars Grunske	Giles Reger	Hongyu Zhang
Esther Guerra	Stephan Reiff-Marganic	
Gabor Guta	Julia Rubin	

Table of Contents

Invited Talk

The Dependability of Complex Socio-technical Systems	1
<i>Ross Anderson</i>	

Automated Learning of Probabilistic Assumptions for Compositional Reasoning	2
<i>Lu Feng, Marta Kwiatkowska, and David Parker</i>	

Verification

An Interface Theory for Service-Oriented Design	18
<i>José Luiz Fiadeiro and Antónia Lopes</i>	

rt-Inconsistency: A New Property for Real-Time Requirements	34
<i>Amalinda Post, Jochen Hoenicke, and Andreas Podelski</i>	

Automatic Flow Analysis for Event-B	50
<i>Jens Bendisposto and Michael Leuschel</i>	

Semantic Quality Attributes for Big-Step Modelling Languages	65
<i>Shahram Esmaeilsabzali and Nancy A. Day</i>	

Specification and Modelling

Formalizing and Operationalizing Industrial Standards	81
<i>Dominik Dietrich, Lutz Schröder, and Ewaryst Schulz</i>	

Modelling Non-linear Crowd Dynamics in Bio-PEPA	96
<i>Mieke Massink, Diego Latella, Andrea Bracciali, and Jane Hillston</i>	

Reachability and Model Checking

Smart Reduction	111
<i>Pepijn Crouzen and Frédéric Lang</i>	

Uniform Monte-Carlo Model Checking	127
<i>Johan Oudinet, Alain Denise, Marie-Claude Gaudel, Richard Lassaigne, and Sylvain Peyronnet</i>	

Model Checking Büchi Pushdown Systems	141
<i>Juncao Li, Fei Xie, Thomas Ball, and Vladimir Levin</i>	

Model Driven Engineering

Modeling with Plausibility Checking: Inspecting Favorable and Critical Signs for Consistency between Control Flow and Functional Behavior	156
<i>Claudia Ermel, Jürgen Gall, Leen Lambers, and Gabriele Taentzer</i>	
Models within Models: Taming Model Complexity Using the Sub-model Lattice	171
<i>Pierre Kelsen, Qin Ma, and Christian Glodt</i>	
Type-Safe Evolution of Spreadsheets	186
<i>Jácome Cunha, Joost Visser, Tiago Alves, and João Saraiva</i>	
A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications	202
<i>Hartmut Ehrig, Claudia Ermel, and Gabriele Taentzer</i>	

Software Development for QoS

A Step-Wise Approach for Integrating QoS throughout Software Development	217
<i>Stéphanie Gatti, Emilie Balland, and Charles Consel</i>	
Systematic Development of UMLsec Design Models Based on Security Requirements	232
<i>Denis Hatebur, Maritta Heisel, Jan Jürjens, and Holger Schmidt</i>	

Testing: Theory and New Trends

Theoretical Aspects of Compositional Symbolic Execution	247
<i>Dries Vanoverberghe and Frank Piessens</i>	
Testing Container Classes: Random or Systematic?	262
<i>Rohan Sharma, Milos Gligoric, Andrea Arcuri, Gordon Fraser, and Darko Marinov</i>	
Seamless Testing for Models and Code	278
<i>Andreas Holzer, Visar Januzaj, Stefan Kugele, Boris Langer, Christian Schallhart, Michael Tautschnig, and Helmut Veith</i>	

Testing in Practice

Retrofitting Unit Tests for Parameterized Unit Testing	294
<i>Suresh Thummala, Madhuri R. Marri, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux</i>	

Evolving a Test Oracle in Black-Box Testing	310
<i>Farn Wang, Jung-Hsuan Wu, Chung-Hao Huang, and Kai-Hsiang Chang</i>	

Automated Driver Generation for Analysis of Web Applications	326
<i>Oksana Tkachuk and Sreeranga Rajan</i>	

On Model-Based Regression Testing of Web-Services Using Dependency Analysis of Visual Contracts	341
<i>Tamim Ahmed Khan and Reiko Heckel</i>	

Code Development and Analysis

Incremental Clone Detection and Elimination for Erlang Programs	356
<i>Huiqing Li and Simon Thompson</i>	

Analyzing Software Updates: Should You Build a Dynamic Updating Infrastructure?	371
<i>Bashar Gharaibeh, Hridesh Rajan, and J. Morris Chang</i>	

Flow-Augmented Call Graph: A New Foundation for Taming API Complexity	386
<i>Qirun Zhang, Wujie Zheng, and Michael R. Lyu</i>	

Search-Based Design Defects Detection by Example	401
<i>Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Manuel Wimmer</i>	

Empirical Studies

An Empirical Study on Evolution of API Documentation	416
<i>Lin Shi, Hao Zhong, Tao Xie, and Mingshu Li</i>	

An Empirical Study of Long-Lived Code Clones	432
<i>Dongxiang Cai and Miryung Kim</i>	

Where the Truth Lies: AOP and Its Impact on Software Modularity	447
<i>Adam Przybytek</i>	

Author Index	463
-------------------------------	------------

The Dependability of Complex Socio-technical Systems

Ross Anderson

Computer Laboratory, University of Cambridge, United Kingdom
Ross.Anderson@cl.cam.ac.uk

Abstract. The story of software engineering has been one of learning to cope with ever greater scale and complexity. We're now building systems with hundreds of millions of users, who belong to millions of firms and dozens of countries; the firms can be competitors and the countries might even be at war.

Rather than having a central planner, we have to arrange things so that the desired behaviour emerges as a result of the self-interested action of many uncoordinated principals. Mechanism design and game theory are becoming as important to the system engineer as more conventional knowledge such as data structures and algorithms. This holds not just for systems no-one really controls, such as the Internet; it extends through systems controlled by small groups of firms, such as the future smart grid, to systems controlled by a single firm, such as Facebook. Once you have hundreds of millions of users, you have to set rules rather than micromanage outcomes.

Other social sciences have a role to play too, especially the behavioural sciences; HCI testing has to be supplemented by a more principled understanding of psychology. And as software comes to pervade just about every aspect of society, software engineers cannot avoid engaging with policy. This has significant implications for academics: for how we educate our students, and for choosing research topics that are most likely to have some impact.

Automated Learning of Probabilistic Assumptions for Compositional Reasoning

Lu Feng, Marta Kwiatkowska, and David Parker

Oxford University Computing Laboratory, Parks Road, Oxford, OX1 3QD, UK

Abstract. Probabilistic verification techniques have been applied to the formal modelling and analysis of a wide range of systems, from communication protocols such as Bluetooth, to nanoscale computing devices, to biological cellular processes. In order to tackle the inherent challenge of scalability, compositional approaches to verification are sorely needed. An example is assume-guarantee reasoning, where each component of a system is analysed independently, using assumptions about the other components that it interacts with. We discuss recent developments in the area of automated compositional verification techniques for probabilistic systems. In particular, we describe techniques to automatically generate probabilistic assumptions that can be used as the basis for compositional reasoning. We do so using algorithmic learning techniques, which have already proved to be successful for the generation of assumptions for compositional verification of non-probabilistic systems. We also present recent improvements and extensions to this work and survey some of the promising potential directions for further research in this area.

1 Introduction

Formal verification is an approach to establishing mathematically rigorous guarantees about the correctness of real-life systems. Particularly successful are fully-automated techniques such as *model checking* [14], which is based on the systematic construction and analysis of a finite-state model capturing the possible states of the system and the transitions between states that can occur over time. Desired properties of the system are formally specified, typically using temporal logic, and checked against the constructed model.

Many real-life systems, however, exhibit *stochastic* behaviour, which analysis techniques must also take into account. For example, components of a system may be prone to failures or messages transmitted between devices may be subjected to delays or be lost. Furthermore, randomisation is a popular tool, for example as a symmetry breaker in wireless communication protocols, or in probabilistic security protocols for anonymity or contract-signing.

Probabilistic verification is a set of techniques for formal modelling and analysis of such systems. *Probabilistic model checking*, for instance, involves the construction of a finite-state model augmented with probabilistic information, such as a Markov chain or probabilistic automaton. This is then checked against properties specified in probabilistic extensions of temporal logic, such as PCTL [25,5].

This permits *quantitative* notions of correctness to be checked, e.g. “the probability of an airbag failing to deploy within 0.02 seconds is at most 0.0001”. It also provides other classes of properties, such as performance or reliability, e.g. “the expected time for a successful transmission of a data packet”.

As with any formal verification technique, one of the principal challenges for probabilistic model checking is *scalability*: the complexity of real-life systems quickly leads to models that are orders of magnitude larger than current verification techniques can support. A promising direction to combat this problem is to adopt *compositional* reasoning methods, which split the work required to verify a system into smaller sub-tasks, based on a decomposition of the system model. A popular approach is the *assume-guarantee* paradigm, in which individual system components are verified under *assumptions* about their environment. Once it has been verified that the other system components do indeed satisfy these assumptions, proof rules can be used to combine individual verification results, establishing correctness properties of the overall system.

In this paper, we discuss some recent developments in the area of automated compositional verification techniques for probabilistic systems, focusing on *assume-guarantee* verification techniques [29,21] for *probabilistic automata* [35,36], a natural model for compositional reasoning. When aiming to develop fully automated verification techniques, an important question that arises is how to devise suitable assumptions about system components. In the context of non-probabilistic systems, a breakthrough in the practical applicability of assume-guarantee verification came about through the adoption of algorithmic *learning* techniques to generate assumptions [15,32]. In recent work [20], we showed how learning could also be successfully used to generate the *probabilistic assumptions* needed for the compositional verification of probabilistic systems. We give an overview of this approach and discuss the relationship with the non-probabilistic case. We also present some recent improvements and extensions to the work and discuss directions for future research.

Paper structure. The rest of the paper is structured as follows. Section 2 gives a summary of probabilistic model checking and probabilistic assume-guarantee reasoning. Section 3 describes learning-based generation of assumptions for non-probabilistic systems, and Section 4 discusses how this can be adapted to the probabilistic case. Section 5 presents experimental results for some further recent developments. Section 6 concludes and identifies areas of future work.

2 Probabilistic Verification

2.1 Modelling and Verification of Probabilistic Systems

We begin by giving a brief overview of automated verification techniques for probabilistic systems, in particular probabilistic model checking. Then, in the next section, we discuss some of the challenges in adding compositional reasoning to these techniques and describe one particular approach to this: a probabilistic assume-guarantee framework.

Probabilistic models. There are a variety of different types of models in common use for probabilistic verification. The simplest are discrete-time Markov chains (DTMCs), whose behaviour is entirely *probabilistic*: every transition in the model is associated with a probability indicating the likelihood of that transition occurring. In many situations, though, it is also important to model *nondeterministic* behaviour. In particular, and crucially for the work described here, this provides a way to capture the behaviour of several parallel components.

Probabilistic automata (PAs) [35,36], and the closely related model of Markov decision processes (MDPs), are common formalisms for modelling systems that exhibit both probabilistic and nondeterministic behaviour. We focus here on PAs, which subsume MDPs and are particularly well suited for compositional reasoning about probabilistic systems [35]. In each state of a PA, several possible actions can be taken and the choice between them is assumed to be resolved in a nondeterministic fashion. Once one of these actions has been selected, the subsequent behaviour is specified by the probability of making a transition to each other state, as for a discrete-time Markov chain. The actions also serve another role: synchronisation. PAs can be composed in parallel, to model the concurrent behaviour of multiple probabilistic processes. In this case, synchronisation between components occurs by taking actions with the same label simultaneously.

Another important aspect that distinguishes probabilistic models is the notion of *time* used. For DTMCs, PAs and MDPs, time is assumed to proceed in discrete steps. In some cases, a more fine-grained model of time may be needed. One possibility is to use continuous-time Markov chains (CTMCs), an extension of DTMCs in which real-valued delays occur between each transition, modelled by exponential distributions. Other, more complex models, which incorporate probabilistic, nondeterministic and real-time behaviour, include probabilistic timed automata (PTAs), continuous-time Markov decision processes (CTMDPs) and interactive Markov chains (IMCs). In this paper, we focus entirely on PAs and thus a discrete model of time. Note, though, that several of the techniques for verifying PTAs reduce the problem to one of analysing a finite state PA; thus, the techniques described in this paper are still potentially applicable.

Probabilistic model checking. The typical approach to specifying properties to be verified on probabilistic systems is to use extensions of temporal logic. The basic idea is to add operators that place a bound on the probability of some event's occurrence. So, whereas in a non-probabilistic setting we might use an LTL formula such as $\square \neg fail$ asserting that, along all executions of the model, *fail* never occurs, in the probabilistic case we might instead use $\langle \square \neg fail \rangle_{\geq 0.98}$. Informally, this means the probability of *fail* never occurring is at least 0.98.

In fact, for models such as probabilistic automata, which exhibit both nondeterministic and probabilistic behaviour, formalising these properties requires care. This is because it is only possible to define the probability of an event's occurrence in the absence of any nondeterminism. The standard approach is to use the notion of *adversaries* (also called strategies, schedulers or policies), which represent one possible way of resolving all nondeterminism in a PA. For an adversary σ , we denote by $Pr_M^\sigma(G)$ the probability of event G when the

nondeterminism in M is resolved by σ , and we say that a PA M satisfies a property $\langle G \rangle_{\geq p_G}$, denoted $M \models \langle G \rangle_{\geq p_G}$, if $Pr_M^\sigma(G) \geq p_G$ for all adversaries σ . Equivalently, M satisfies $\langle G \rangle_{\geq p_G}$ when $Pr_M^{\min}(G) \geq p_G$, where $Pr_M^{\min}(G)$ denotes the minimum probability, over all adversaries, of G .

A useful class of properties for PAs are *probabilistic safety properties*. These take the form $\langle G \rangle_{\geq p_G}$, as described above, where G is a *safety property* (a set of “good” model traces, defined by a set of “bad” prefixes, finite traces for which any extension is *not* a “good” trace). More precisely, we assume here that the “bad” traces form a regular language and, for efficiency, we represent this using a deterministic finite automaton (DFA), denoted G^{err} , rather than, say, temporal logic. Probabilistic safety properties can capture a wide range of useful properties of probabilistic automata, including for example:

- “event A always occurs before event B with probability at least 0.9”
- “the probability of a system failure occurring is at most 0.02”
- “the probability of terminating within k time-units is at least 0.75”

Many other classes of properties are also in common use for probabilistic verification. First, we can generalise $\langle G \rangle_{\geq p_G}$ to $\langle \phi \rangle_{\sim p}$ where $\sim \in \{<, \leq, \geq, >\}$ and ϕ is any ω -regular language (subsuming, for example, the temporal logic LTL). We can also consider *branching-time* probabilistic temporal logics such as PCTL or PCTL* [25,5], as opposed to the *linear-time* properties considered so far. Yet another possibility is to augment the PA with costs or rewards and consider, for example, the expected total cumulated reward or the expected long-run average reward. Finally, we mention *multi-objective* properties, which can be used to express trade-offs between multiple quantitative (e.g. probabilistic linear-time) properties across the set of adversaries of a PA. Multi-objective probabilistic model checking (see e.g. [18,21]) is a key ingredient in the implementation of the probabilistic assume-guarantee framework discussed in this paper.

Algorithmically, probabilistic model checking for PAs reduces in most cases to a combination of graph-based algorithms and numerical computation [16,1]. For the latter, either approximate iterative calculations (e.g. value iteration) can be used or a reduction to linear programming (LP). Prior to this, it is often necessary to construct a *product* PA for analysis. For example, for the probabilistic safety properties described above, we would construct the synchronous product of the PA to be verified and the DFA representing the safety property [29]. Tool support for verifying PAs (or MDPs) is also available: several probabilistic model checkers have been developed and are widely used. The most popular of these is PRISM [26]; others include LiQuor [13], RAPTURE [27] and ProbDiVinE [4].

Example 1. In Figure 1, we show a simple example (taken from [29]) comprising two components, each modelled as a PA. Component M_1 represents a controller that powers down devices. When it identifies a problem (modelled by the action *detect*), it sends two messages, represented by actions *warn* and *shutdown*, respectively. However, with probability 0.2 it will fail to issue the *warn* message first. The second component, M_2 represents a device to be powered down by M_1 . It expects to receive two messages, *warn* and *shutdown*. If the first of these is absent, it will only shut down correctly 90% of the time.

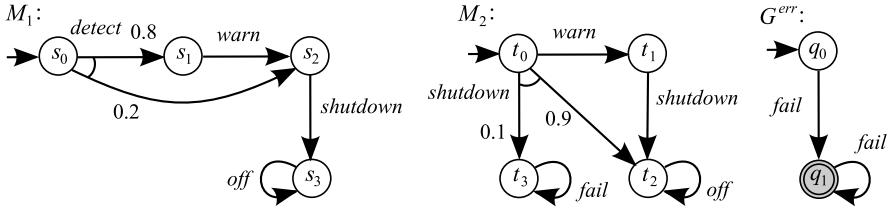


Fig. 1. Example, from [29]: two PAs M_1, M_2 and the DFA G^{err} for a safety property G ; we have that $M_1 \parallel M_2$ satisfies probabilistic safety property $\langle G \rangle_{\geq 0.98}$

We consider the parallel composition $M_1 \parallel M_2$ of the two devices and check a simple probabilistic safety property: “with probability 0.98, action *fail* never occurs”. Formally, this is captured as $\langle G \rangle_{\geq 0.98}$, where G is the safety property “action *fail* never occurs”, represented by the DFA G^{err} also shown in Figure 1 (this is over the alphabet $\{\text{fail}\}$), so any trace apart from the empty string is accepted as an error trace). It can be seen that the maximum probability of *fail* occurring in $M_1 \parallel M_2$ is $0.2 \cdot 0.1 = 0.02$. So, we have that $M_1 \parallel M_2 \models \langle G \rangle_{\geq 0.98}$.

2.2 Compositional Reasoning for Probabilistic Systems

Verification techniques for probabilistic systems can be expensive, both in terms of their time and space requirements, making scalability an important challenge. A promising way to address this is to perform verification in a *compositional* manner, decomposing the problem into sub-tasks which analyse each component separately. However, devising compositional analysis techniques for probabilistic systems requires considerable care, particularly for models such as probabilistic automata that exhibit both probabilistic and nondeterministic behaviour [35].

Assume-guarantee reasoning. In the case of *non-probabilistic* models, a popular approach to compositional verification is the use of *assume-guarantee* reasoning. This is based on checking assume-guarantee *triples* of the form $\langle A \rangle M_i \langle G \rangle$, with the meaning “whenever component M_i is part of a system satisfying the *assumption* A , then the system is *guaranteed* to satisfy property G ”. Proof rules can then be established that combine properties of individual components to show correctness properties of the overall system.

To give a concrete example, we adopt the framework used in [15,32]. Components M_i and assumption A are labelled transition systems and G is a safety property. Then, $\langle A \rangle M_i \langle G \rangle$ has the meaning that $A \parallel M_i \models G$. We also say that a component M_j satisfies an assumption A if all traces of M_j are included in A , denoted $M_j \sqsubseteq A$. For components M_1 and M_2 , the following proof rule holds:

$$\frac{\begin{array}{c} M_1 \sqsubseteq A \\ \langle A \rangle M_2 \langle G \rangle \end{array}}{M_1 \parallel M_2 \models G}$$

Thus, verifying property G on the combined system $M_1 \parallel M_2$ reduces to two separate (and hopefully simpler) checks, one on M_1 and one on $A \parallel M_2$.

Assume-guarantee for probabilistic systems. There are several ways that we could attempt to adapt the above assume-guarantee framework to probabilistic automata. A key step is to formalise the notion of an assumption A about a (probabilistic) component M_i . The two most important requirements are: (i) that it is *compositional*, allowing proof rules such as the one above to be constructed; and (ii) that the premises of the proof rule can be checked *efficiently*.

Unfortunately, the most natural probabilistic extension of the trace inclusion preorder \sqsubseteq used above, namely *trace distribution inclusion* [35] is *not* compositional [34]. One way to address this limitation is to restrict the ways in which components can be composed in parallel; examples include the switched probabilistic I/O automata model of [12] and the synchronous parallel composition of probabilistic Reactive Modules used in [2]. Another is to characterise variants of the trace distribution inclusion preorder that *are* compositional, e.g. [35,30]. However, none of these preorders can be checked efficiently, limiting their applicability to automated compositional verification.

Other candidates for ways to formalise the relationship between a component PA and its assumption include the notions of (strong and weak) probabilistic simulation and bisimulation [35,36]. There are a number of variants, most of which are compositional. Unfortunately, for the weak variants, efficient methods to check the relations are not yet known and, for the strong variants, although relatively efficient algorithms exist, the relations are usually too coarse to yield suitably small assumptions. Other work in this area includes [17], which uses a notion of probabilistic contracts to reason compositionally about systems with both stochastic and nondeterministic behaviour. Automating the techniques in an implementation has not yet been attempted.

In this paper, we focus on the probabilistic assume-guarantee framework of [29]. This makes no restrictions on the PAs that can be analysed, nor the way that they are composed in parallel; instead it opts to use a less expressive form for assumptions, namely probabilistic safety properties. An assume-guarantee triple now takes the form $\langle A \rangle_{\geq p_A} M_i \langle G \rangle_{\geq p_G}$ where M_i is a PA and $\langle A \rangle_{\geq p_A}, \langle G \rangle_{\geq p_G}$ are two probabilistic safety properties; the former is a *probabilistic assumption*, the latter a property to be checked. This is interpreted as follows: $\langle A \rangle_{\geq p_A} M_i \langle G \rangle_{\geq p_G}$ is true if, for all adversaries σ of M_i such that $Pr_{M_i}^\sigma(A) \geq p_A$ holds, $Pr_{M_i}^\sigma(G) \geq p_G$ also holds. Crucially, verifying whether this is true reduces to a multi-objective model checking problem [18,29], which can be carried out efficiently by solving an LP problem. The proof rule used earlier now becomes:

$$\frac{M_1 \models \langle A \rangle_{\geq p_A} \quad \langle A \rangle_{\geq p_A} M_2 \langle G \rangle_{\geq p_G}}{M_1 \parallel M_2 \models \langle G \rangle_{\geq p_G}} \quad (\text{ASYM})$$

thus reducing the problem of checking property $\langle G \rangle_{\geq p_G}$ on $M_1 \parallel M_2$ to two smaller sub-problems: (i) verifying a probabilistic safety property on M_1 ; and (ii) checking an assume-guarantee triple for M_2 .

In [29], this proof rule, along with several others, are proved and then used to perform compositional verification on a set of large case studies. This includes

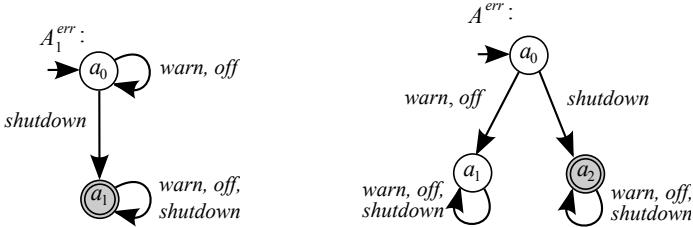


Fig. 2. DFAs for two learning-generated assumptions to verify the system from Figure 1 compositionally with rule (ASYM); A (right) is a valid assumption, A_1 (left) is not

cases where non-compositional verification is either slower or infeasible. In recent work [21], this framework is extended to permit the use of more expressive assumptions (and properties): *quantitative multi-objective properties*, essentially Boolean combinations of probabilistic safety, ω -regular and expected total reward properties. An example of an expressible assumption is “with probability 1, component M_i always eventually sends a message and the expected time to do so is at most 10 time-units.” Again, the framework is implemented through multi-objective model checking and applied to several case studies.

Example 2. Recall the PAs M_1, M_2 and probabilistic safety property $\langle G \rangle_{\geq 0.98}$ from Example 1, for which we stated that $M_1 \parallel M_2 \models \langle G \rangle_{\geq 0.98}$. This property can also be checked in a compositional manner using the rule (ASYM). We use an assumption about the behaviour of M_1 which can be stated informally as “with probability at least 0.8, *warn* occurs before *shutdown*”. This takes the form of a probabilistic safety property $\langle A \rangle_{\geq 0.8}$ where the DFA for A is shown on the right-hand side of Figure 2. To perform compositional verification, we perform two separate checks, $M_1 \models \langle A \rangle_{\geq 0.8}$ and $\langle A \rangle_{\geq 0.8} M_2 \langle G \rangle_{\geq 0.98}$, both of which hold.

3 Learning Assumptions for Compositional Verification

The ideas behind assume-guarantee verification for non-probabilistic systems have a long history [28]. Making these techniques work in practice, however, is a challenge. In particular, deciding how to break down a system into its components and devising suitable assumptions about the behaviour of those components initially proved difficult to automate. A breakthrough in this area came with the observation of [15] that *learning* techniques, such as Angluin’s L* algorithm [3], could be used to automate the process of generating assumptions. In this section, we give a short description of L* and its application to automatic compositional verification of non-probabilistic systems.

The L* algorithm. L* [3] is a learning algorithm for generating a *minimal* DFA that accepts an unknown regular language \mathcal{L} . It uses the *active* learning model, interacting with a *teacher* to which it can pose *queries*. There are two kinds of these: *membership queries*, asking whether a particular word t is contained in \mathcal{L} ; and *equivalence queries*, asking whether a conjectured automaton

A accepts exactly \mathcal{L} . Initially, L^* poses a series of membership queries to the teacher, maintaining results in an *observation table*. After some time (more precisely, when the table is *closed* and *consistent*; see [3] for details), L^* generates a conjecture A and submits this as an equivalence query. If the answer to this query is “no”, the teacher must provide a counterexample c , in either positive form ($c \in \mathcal{L} \setminus \mathcal{L}(A)$) or negative form ($c \in \mathcal{L}(A) \setminus \mathcal{L}$). The algorithm then resumes submission of membership queries until the next conjecture can be generated.

The total number of queries required by L^* is polynomial in the size of the minimal DFA required to represent the language \mathcal{L} being learnt. Furthermore, the number of equivalence queries is bounded by the number of states in the automaton since at most one new state is added at each iteration.

Learning assumptions with L^* . In [15], it was shown how L^* could be adapted to the problem of automatically generating assumptions for assume-guarantee verification of labelled transition systems. This is done by phrasing the problem in a language-theoretic setting: the assumption A to be generated is a labelled transition system, whose set of (finite) traces forms a regular language.

The approach works by using the notion of the *weakest assumption* [22] as the target language \mathcal{L} . Intuitively, assuming the non-probabilistic proof rule from Section 2.2, the weakest assumption is the set of all possible traces of a process that, when put in parallel with M_2 , do not violate the property G . Thus, the membership query used by the teacher checks, for a trace t , whether $t \parallel M_2 \models G$, where t denotes a transition system comprising the single trace t . Both this and the required equivalence queries can be executed automatically by a model checker. An important observation, however, is that, in practice, this is not usually the language actually learnt. The algorithm works in such a way that conjectured assumptions generated as L^* progresses may be sufficient to either prove or refute the property G , allowing the procedure to terminate early.

A variety of subsequent improvements and extensions to the basic technique of [15] were proposed (see e.g. [32] for details) and the approach was successfully applied to several large case studies. It tends to perform particularly well when the size of a generated assumption and the size of its alphabet remain small. The work has sparked a significant amount of interest in the area in recent years. There have been several attempts to improve performance, including symbolic implementations [31] and optimisations to the use of L^* [9]. Others have also devised alternative learning-based methods, for example by reformulating the assumption generation problem as one of computing the smallest finite automaton separating two regular languages [23,11], or using the CDNF learning algorithm to generate implicit representations of assumptions [10].

4 Learning Probabilistic Assumptions

As mentioned in Section 2.2, there are several different possibilities for the type of assumption used to perform probabilistic assume-guarantee verification. We focus on the use of probabilistic safety properties, as in [29]. Although these

have limited expressivity, an advantage is that the generation of such assumptions can be automated by adapting the L*-based techniques developed for non-probabilistic compositional verification [15,32].

This approach was proposed in [20] and shown to be applicable on several large case studies. In this section, we give a high-level overview of the approach and describe the key underlying ideas; for the technical details, the reader is referred to [20]. The basic setting is as outlined in Section 2.2: we consider the assume-guarantee proof rule (ASYM) from [29], applied to check that the parallel composition of two PAs M_1 and M_2 satisfies a property $\langle G \rangle_{\geq p_G}$. To do this, we need an assumption $\langle A \rangle_{\geq p_A}$, which will be generated automatically.

Probabilistic and non-probabilistic assumptions. The first key point to make is that, although we are required to learn a *probabilistic assumption*, i.e. a probabilistic safety property $\langle A \rangle_{\geq p_A}$, we can essentially reduce this task to the problem of learning a *non-probabilistic* assumption, i.e. the corresponding safety property A . The reasoning behind this is as follows.

We need the probabilistic assumption $\langle A \rangle_{\geq p_A}$ to be such that both premises of the proof rule (ASYM) hold: (i) $M_1 \models \langle A \rangle_{\geq p_A}$; and (ii) $\langle A \rangle_{\geq p_A} M_2 \langle G \rangle_{\geq p_G}$. However, if (i) holds for a particular value of p_A , then it also holds for any lower value of p_A . Conversely, if (ii) holds for some p_A , it then must hold for any higher value. Thus, given a safety property A , we can determine an appropriate probability bound p_A (if one exists) by finding the lowest value of p_A (if any) such that (ii) holds and checking it against (i). Alternatively, we can find the highest value of p_A such that (i) holds (this is just $Pr_{M_1}^{\min}(A)$ in fact) and seeing if this suffices for (ii). A benefit of the latter is that, even if $M_1 \parallel M_2 \models \langle G \rangle_{\geq p_G}$ cannot be shown to be true with this particular assumption, we still obtain a *lower bound* on $Pr_{M_1 \parallel M_2}^{\min}(G)$. Furthermore, with an additional simple check, an *upper bound* can also be generated (see [20] for details).

Adapting the L* algorithm. Next, we describe how we adapt the L*-based approach of [15,32] for generating non-probabilistic assumptions to our setting. The underlying idea behind the use of L* in [15,32] is the notion of *weakest assumption*: this will always exist and, if the property G being verified is true, will permit a compositional verification. It is used as the target language for L* and forms the basis of the membership and equivalence queries. In practice, however, this language is often not the one that is finally generated since intermediate conjectured assumptions may suffice, either to show that the property is true or that it is false.

An important difference in the probabilistic assume-guarantee framework of [29] is that it is *incomplete*, meaning that, even if the property $\langle G \rangle_{\geq p_G}$ holds, there may be no probabilistic assumption $\langle A \rangle_{\geq p_A}$ for which the rule (ASYM) can be used to prove the property correct. So, there can be no equivalent notion of weakest assumption to be used as a target language. However, we can adopt a similar approach whereby we use L* to generate a sequence of conjectured assumptions A and, for each one, potentially show that $\langle G \rangle_{\geq p_G}$ is either true or false. Furthermore, as described above, each assumption A yields a lower and

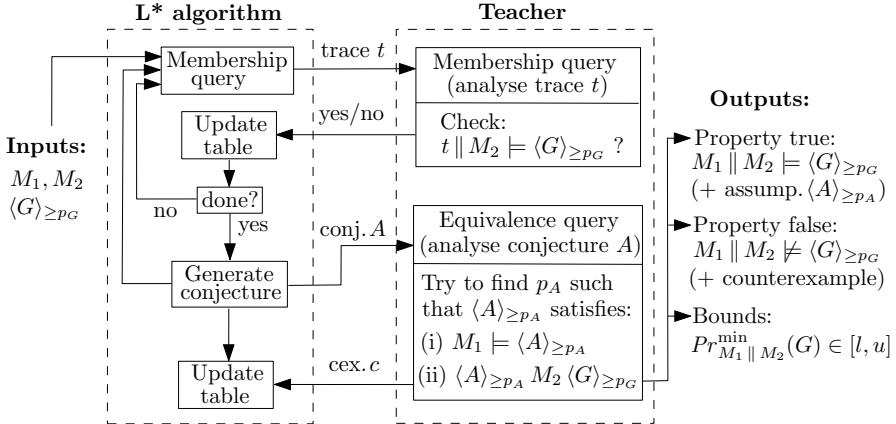


Fig. 3. Overview of probabilistic assumption generation [20], using an adaption of L*: generates assumption $\langle A \rangle_{\geq p_A}$ for verification of $M_1 \parallel M_2 \models \langle G \rangle_{\geq p_G}$ using rule (ASYM)

an upper bound on $Pr_{M_1 \parallel M_2}^{\min}(G)$. We can retain these values as the learning algorithm progresses, keeping the highest lower bound and lowest upper bound discovered to report back to the user.¹ This means that, even if the algorithm terminates early (without concluding that $\langle G \rangle_{\geq p_G}$ is true or false), it produces useful *quantitative* information about the property of interest.

In order to produce conjectures, L* needs answers to membership queries about whether certain traces t should be in the language being generated. For this, we use the following check: $t \parallel M_2 \models \langle G \rangle_{\geq p_G}$, which can be seen as an analogue of the corresponding one for the non-probabilistic case. Intuitively, the idea is that if, under a single possible behaviour t of M_1 , M_2 satisfies the property, then t should be included in the assumption A . There may be situations where this scheme leads to an assumption that cannot be used to verify the property. This is because it is possible that there are multiple traces which do not violate property $\langle G \rangle_{\geq p_G}$ individually but, when combined, cause $\langle G \rangle_{\geq p_G}$ to be false. In practice, though, this approach seems to work well in most cases.

A final aspect of L* that needs discussion is *counterexamples*. In [15,32], when the response to an equivalence query is “no”, a counterexample in the form of a trace is returned to L*. In our case, there are two differences in this respect. Firstly, a counterexample may constitute multiple traces; this is because the results of the model checking queries executed by the teacher yield *probabilistic counterexamples* [24], which comprise multiple paths (again, see [20] for precise details). Secondly, situations may arise where no such counterexample can be generated (recall that there is no guarantee that an assumption can eventually be created). In this case, since no trace can be returned to L*, we terminate the learning algorithm, returning the current tightest bounds on $Pr_{M_1 \parallel M_2}^{\min}(G)$ that have been computed so far.

¹ Note that the sequence of assumptions generated is *not* monotonic, e.g. it does *not* yield a sequence of increasing lower bounds on $Pr_{M_1 \parallel M_2}^{\min}(G)$.

The learning loop. We summarise in Figure 3 the overall structure of the L*-based algorithm for generating probabilistic assumptions. The left-hand side shows the basic L* algorithm. This interacts, through queries, with the teacher, shown on the right-hand side. The teacher responds to queries as described above. Notice that the equivalence query, which analyses a particular conjectured assumption A , has four possible outcomes: the first two are when A can be used to show either that $M_1 \parallel M_2 \models \langle G \rangle_{\geq p_G}$ or $M_1 \parallel M_2 \not\models \langle G \rangle_{\geq p_G}$; the third is when a counterexample (comprising one or more traces) is passed back to L*; and the fourth is when no counterexample can be generated so the algorithm returns the tightest bounds computed so far for $\Pr_{M_1 \parallel M_2}^{\min}(G)$.

Example 3. Figure 2 shows the two successive conjunction assumptions generated by the learning loop, when applied to the PAs and property of Example 1. The first conjecture A_1 does not permit (ASYM) to be applied but a counterexample can be found. L* then generates the conjecture A , which we know from Example 2 *does* allow compositional verification that $M_1 \parallel M_2 \models \langle G \rangle_{\geq p_G}$.

5 Experimental Results

The approach outlined in the previous section was successfully used in [20] to automatically generate probabilistic assumptions for several large case studies. Furthermore, these assumptions were much smaller than the corresponding components that they represented, leading to gains in performance. In this section, we present some recent extensions and improvements to that work.

5.1 A New Case Study: Mars Exploration Rovers

First, we present an application of the techniques to a new case study, based on a module from the flight software for JPL’s Mars Exploration Rovers (MER). Compositional verification of a non-probabilistic model of this system was performed previously in [33]. The module studied is a *resource arbiter*, which controls access to various shared resources between a set of user threads, each of which performs a different application on the rover.

The arbiter also enforces priorities between resources, granting and rescinding access rights to users as required. For example, it is considered that communication is more important than driving; so, if a communication request is received while the rover is driving, the arbiter will rescind permission to use the drive motors in order to grant permission for use of the rover’s antennas.

Our model adds the possibility of faulty behaviour. When the arbiter sends a *rescind* message to a user thread, there is a small chance of the message being lost in transmission. We also add information about the likelihood of a user thread requesting a given type of resource in each cycle of the system’s execution. We are interested in a mutual exclusion property which checks that permission for communication and driving is not granted simultaneously by the arbiter. More precisely, we verify a probabilistic safety property relating to “the minimum probability that mutual exclusion is not violated within k cycles of

Case study [parameters]		Component sizes		Compositional (L*)				Compositional (NL*)				Non-comp.
		$ M_2 \otimes G^{err} $	$ M_1 $	$ A^{err} $	MQ	EQ	Time	$ A^{err} $	MQ	EQ	Time	Time
<i>client-server1</i>	3	81	16	5	99	3	6.8	6	223	4	7.7	0.02
	5	613	36	7	465	5	21.6	8	884	5	26.1	0.04
	7	4,733	64	9	1,295	7	484.6	10	1,975	5	405.9	0.08
<i>client-serverN</i>	3	229	16	5	99	3	6.6	6	192	3	7.4	0.04
	4	1,121	25	6	236	4	26.1	7	507	4	33.1	0.12
	5	5,397	36	7	465	5	191.1	8	957	5	201.9	0.28
<i>consensus</i>	2 3 20	391	3,217	6	149	5	24.2	7	161	3	14.9	108.1
	2 4 4	573	431,649	12	2,117	8	413.2	12	1,372	5	103.4	2.59
	3 3 20	8,843	38,193	11	471	6	438.9	15	1,231	5	411.3	>24h
<i>sensor network</i>	1	42	72	3	17	2	3.5	4	31	2	3.9	0.03
	2	42	1,184	3	17	2	3.7	4	31	2	4.0	0.25
	3	42	10,662	3	17	2	4.6	4	31	2	4.8	2.01
<i>mer</i>	2 2	961	85	4	113	3	9.0	7	1,107	5	28.9	0.09
	2 5	5,776	427,363	4	113	3	31.8	7	1,257	5	154.4	1.96
	3 2	16,759	171	4	173	3	210.5	—	—	—	mem-out	0.42

Fig. 4. Performance comparison of the L*- and NL*-based methods for rule (ASYM)

system execution”. The model comprises N user threads U_1, \dots, U_N and the arbiter ARB which controls R shared resources (in the full system model, N is 11 and R is 15). We perform compositional verification using the rule (ASYM), decomposing the system into two parts: $M_1 = U_1 \| U_2 \| \dots \| U_N$ and $M_2 = ARB$.

5.2 A Comparison of Learning Methods: L* versus NL*

Next, we investigate the use of an alternative learning algorithm to generate probabilistic assumptions. Whereas L* learns a minimal DFA for a regular language, the algorithm NL* [7] learns a minimal *residual finite-state automaton* (RFSA). RFSA are a subclass of nondeterministic finite automata. For the same regular language \mathcal{L} , the minimal RFSA that accepts \mathcal{L} can be exponentially more succinct than the corresponding minimal DFA. In fact, for the purposes of probabilistic model checking, the RFSA needs to be determinised anyway [1]. However, the hope is that the smaller size of the RFSA may lead to a faster learning procedure. NL* works in a similar fashion to L*, making it straightforward to substitute into the learning loop shown in Figure 3.

We added NL* to our existing implementation from [20], which is based on an extension of PRISM [26] and the `libalif` [6] learning library. We then compared the performance of the two learning algorithms on a set of five case studies. The first four are taken from [20]: client-server benchmark models from [32] incorporating failures in one or all clients (*client-server1* and *client-serverN*), Aspnes & Herlihy’s randomised consensus algorithm (*consensus*) and a sensor network exhibiting message losses (*sensor network*). The fifth example is the MER model from above (*mer*). Experiments were run on a 1.86GHz PC with 2GB RAM and we imposed a time-out of 24 hours.

Figure 4 compares the performance of the L*-based and NL*-based methods to generate probabilistic assumptions for the rule (ASYM). The “Component sizes” columns give the state space of the two components, M_1 and M_2 , in each model; for M_2 , this also includes the automaton for the safety property

Case study [parameters]	Component sizes		(ASYM)		(ASYM-N)		Non-comp.
	$ M_2 \otimes G^{err} $	$ M_1 $	$ A^{err} $	Time (s)	$ A^{err} $	Time (s)	Time (s)
<i>client-serverN</i> [N]	6 7	25,801 123,053	49 64	— — mem-out mem-out	8 24	40.9 164.7	0.7 1.7
<i>mer</i> [N R]	3 5 4 5 5 5	224,974 7,949,992 265,559,722	1,949,541 6,485,603 17,579,131	— — mem-out mem-out	4 4 4	29.8 122.9 3,903.4	48.2 mem-out mem-out

Fig. 5. Performance comparison of the rule (ASYM) and the rule (ASYM-N)

G being checked. For each method, we report the size of the learnt assumption A^{err} (DFA or RFSA), the number of membership queries (MQ) and equivalence queries (EQ) needed, and the total time (in seconds) for learning. We also give the time for non-compositional verification using PRISM.

With the exception of one model, both algorithms successfully generated a correct (and small) assumption in all cases. The results show that the L^* -based method is faster than NL^* in most cases. However, on several of the larger models, NL^* has better performance due to a smaller number of equivalence queries. NL^* needs more membership queries, but these are less costly. We do not compare the execution time of our prototype tool with the (highly-optimised) PRISM in detail. But, it is worth noting that, for two of the *consensus* models, compositional verification is actually faster than non-compositional verification.

5.3 Learning Multiple Assumptions: Rule (ASYM-N)

Lastly, in this section, we consider an extension of the probabilistic assumption generation scheme of Section 4, adapting it to the proof rule (ASYM-N) of [29]. This is motivated by the observation that, for several of the case studies in the previous section, scalability is limited because one of the two components comprises several sub-components. As the number of sub-components increases, model checking becomes infeasible due to the size of the state space. The (ASYM) proof rule allows decomposition of the system into more than 2 components:

$$\begin{array}{c}
 \langle \text{true} \rangle M_1 \langle A_1 \rangle_{\geq p_1} \\
 \langle A_1 \rangle_{\geq p_1} M_2 \langle A_2 \rangle_{\geq p_2} \\
 \dots \\
 \langle A_{n-1} \rangle_{\geq p_{n-1}} M_n \langle G \rangle_{\geq p_G} \\
 \hline
 \langle \text{true} \rangle M_1 \parallel \dots \parallel M_n \langle G \rangle_{\geq p_G}
 \end{array} \quad (\text{ASYM-N})$$

For the earlier MER case study, for instance, we can now decompose the system into $N+1$ components: the N user threads U_1, U_2, \dots, U_N and the arbiter ARB .

Adapting our probabilistic assumption generation process to (ASYM-N) works by learning assumptions for the rule in a recursive fashion, with each step requiring a separate instantiation of the learning algorithm for (ASYM), as is done for the non-probabilistic version of a similar rule in [32]. Experimental results are presented in Figure 5 for two of the case studies from Section 5.2. The results

demonstrate that, using the rule (ASYM-N), we can successfully learn small assumptions and perform compositional verification in several cases where the rule (ASYM) runs out of memory. Furthermore, in two instances, (ASYM-N) permits verification of models which cannot be checked in a non-compositional fashion.

6 Conclusions and Future Work

We have discussed recent progress in the development of automated compositional verification techniques for probabilistic systems, focusing on the assume-guarantee framework of [29,21] for probabilistic automata. We also described how the verification process can be automated further using learning-based generation of the assumptions needed to apply assume-guarantee proof rules and described some recent improvements and extensions to this work.

There are a variety of possible directions for future research in this area. One is to extend our techniques for learning probabilistic assumptions to the assume-guarantee framework in [21], which additionally includes ω -regular and expected reward properties. Here, the ω -regular language learning algorithms of [19,8] may provide a useful starting point. There are also possibilities to enhance the underlying compositional verification framework. This includes developing efficient techniques to work with richer classes of probabilistic assumption and extending the approach to handle more expressive types of probabilistic models, such as those that incorporate continuous-time behaviour.

Acknowledgments. The authors are part supported by ERC Advanced Grant VERIWARE, EU FP7 project CONNECT and EPSRC grant EP/F001096/1.

References

1. de Alfaro, L.: Formal Verification of Probabilistic Systems. Ph.D. thesis, Stanford University (1997)
2. de Alfaro, L., Henzinger, T.A., Jhala, R.: Compositional methods for probabilistic systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, p. 351. Springer, Heidelberg (2001)
3. Angluin, D.: Learning regular sets from queries and counterexamples. *Information and Computation* 75(2), 87–106 (1987)
4. Barnat, J., Brim, L., Cerna, I., Ceska, M., Tumova, J.: ProbDiVinE-MC: Multi-core LTL model checker for probabilistic systems. In: Proc. QEST 2008 (2008)
5. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995)
6. Bollig, B., Katoen, J.-P., Kern, C., Leucker, M., Neider, D., Piegdon, D.R.: libalf: The automata learning framework. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 360–364. Springer, Heidelberg (2010)
7. Bollig, B., Habermehl, P., Kern, C., Leucker, M.: Angluin-style learning of NFA. In: Proc. IJCAI 2009, pp. 1004–1009. Morgan Kaufmann Publishers Inc., San Francisco (2009)

8. Chaki, S., Gurfinkel, A.: Automated assume-guarantee reasoning for omega-regular systems and specifications. In: Proc. NFM 2010, pp. 57–66 (2010)
9. Chaki, S., Strichman, O.: Optimized L*-based assume-guarantee reasoning. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 276–291. Springer, Heidelberg (2007)
10. Chen, Y.-F., Clarke, E.M., Farzan, A., Tsai, M.-H., Tsay, Y.-K., Wang, B.-Y.: Automated assume-guarantee reasoning through implicit learning. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 511–526. Springer, Heidelberg (2010)
11. Chen, Y.-F., Farzan, A., Clarke, E.M., Tsay, Y.-K., Wang, B.-Y.: Learning minimal separating dFA's for compositional verification. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 31–45. Springer, Heidelberg (2009)
12. Cheung, L., Lynch, N.A., Segala, R., Vaandrager, F.W.: Switched probabilistic I/O automata. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 494–510. Springer, Heidelberg (2005)
13. Ciesinski, F., Baier, C.: Liquor: A tool for qualitative and quantitative linear time analysis of reactive systems. In: Proc. QEST 2006, pp. 131–132. IEEE CS Press, Los Alamitos (2006)
14. Clarke, E., Grumberg, O., Peled, D.: Model Checking. The MIT Press, Cambridge (2000)
15. Cobleigh, J., Giannakopoulou, D., Pasareanu, C.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
16. Courcoubetis, C., Yannakakis, M.: Markov decision processes and regular events. *IEEE Transactions on Automatic Control* 43(10), 1399–1418 (1998)
17. Delahaye, B., Caillaud, B., Legay, A.: Probabilistic contracts: A compositional reasoning methodology for the design of stochastic systems. In: Proc. ACSD 2010, pp. 223–232. IEEE CS Press, Los Alamitos (2010)
18. Etessami, K., Kwiatkowska, M., Vardi, M., Yannakakis, M.: Multi-objective model checking of Markov decision processes. *LMCS* 4(4), 1–21 (2008)
19. Farzan, A., Chen, Y.-F., Clarke, E.M., Tsay, Y.-K., Wang, B.-Y.: Extending automated compositional verification to the full class of omega-regular languages. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 2–17. Springer, Heidelberg (2008)
20. Feng, L., Kwiatkowska, M., Parker, D.: Compositional verification of probabilistic systems using learning. In: Proc. 7th International Conference on Quantitative Evaluation of Systems (QEST 2010), pp. 133–142. IEEE CS Press, Los Alamitos (2010)
21. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Quantitative multi-objective verification for probabilistic systems. In: Abdulla, P.A. (ed.) TACAS 2011. LNCS, vol. 6605, pp. 112–127. Springer, Heidelberg (2011)
22. Giannakopoulou, D., Pasareanu, C., Barringer, H.: Component verification with automatically generated assumptions. *ASE* 12(3), 297–320 (2005)
23. Gupta, A., McMillan, K., Fu, Z.: Automated assumption generation for compositional verification. *Formal Methods in System Design* 32(3), 285–301 (2008)
24. Han, T., Katoen, J.P., Damman, B.: Counterexample generation in probabilistic model checking. *IEEE Transactions on Software Engineering* 35(2), 241–257 (2009)
25. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. *Formal Aspects of Computing* 6(5), 512–535 (1994)

26. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
27. Jeannet, B., D’Argenio, P., Larsen, K.: Rapture: A tool for verifying Markov decision processes. In: Proc. CONCUR 2002 Tools Day, pp. 84–98 (2002)
28. Jones, C.: Tentative steps towards a development method for interfering programs. ACM Transactions on Programming Languages and Systems 5(4), 596–619 (1983)
29. Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Assume-guarantee verification for probabilistic systems. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 23–37. Springer, Heidelberg (2010)
30. Lynch, N., Segala, R., Vaandrager, F.: Observing branching structure through probabilistic contexts. SIAM Journal on Computing 37(4) (2007)
31. Nam, W., Madhusudan, P., Alur, R.: Automatic symbolic compositional verification by learning assumptions. FMSD 32(3), 207–234 (2008)
32. Pasareanu, C., Giannakopoulou, D., Bobaru, M., Cobleigh, J., Barringer, H.: Learning to divide and conquer: Applying the L* algorithm to automate assume-guarantee reasoning. Formal Methods in System Design 32(3), 175–205 (2008)
33. Pasareanu, C., Giannakopoulou, D.: Towards a compositional SPIN. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 234–251. Springer, Heidelberg (2006)
34. Segala, R.: A compositional trace-based semantics for probabilistic automata. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 234–248. Springer, Heidelberg (1995)
35. Segala, R.: Modelling and Verification of Randomized Distributed Real Time Systems. Ph.D. thesis, Massachusetts Institute of Technology (1995)
36. Segala, R., Lynch, N.: Probabilistic simulations for probabilistic processes. Nordic Journal of Computing 2(2), 250–273 (1995)

An Interface Theory for Service-Oriented Design

José Luiz Fiadeiro¹ and Antónia Lopes²

¹ Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
jose@mcs.le.ac.uk
² Faculty of Sciences, University of Lisbon
Campo Grande, 1749–016 Lisboa, Portugal
mal@di.fc.ul.pt

Abstract. We revisit the notions of interface and component algebra proposed by de Alfaro and Henzinger in [7] for component-based design and put forward elements of a corresponding interface theory for service-oriented design. We view services as a layer that can be added over a component infrastructure and propose a notion of service interface for a component algebra that is an asynchronous version of relational nets adapted to SCA (the Service Component Architecture developed by the Open Service-Oriented Architecture collaboration).

1 Services vs. Components, Informally

In [7], de Alfaro and Henzinger put forward a number of important insights, backed up by mathematical models, that led to an abstract characterisation of essential aspects of component-based software design (CBD), namely in the distinction between the notions of component and interface, and the way they relate to each other. In this paper, we take stock on the work that we developed in the FET-GC2 integrated project SENSORIA [21] towards a language and mathematical model for service-oriented modelling [12], and investigate what abstractions can be put forward for service-oriented computing (SOC) that relate to the notions of interface and component algebra proposed in [7]. Our ultimate goal is similar to that of [7]: to characterise the fundamental structures that support SOC independently of the specific formalisms (Petri-nets, automata, process calculi, *inter alia*) that may be adopted to provide models for languages or tools.

A question that, in this context, cannot be avoided, concerns the difference between component-based and service-oriented design. The view that we adopt herein is that, on the one hand, services offer a layer of activity that can be superposed over a component infrastructure (what is sometimes referred to as a ‘service overlay’) and, on the other hand, the nature of the interactions between processes that is needed to support such a service overlay is intrinsically asynchronous and conversational, which requires a notion of component algebra that is different from the ones investigated in [7] for CBD.

The difference between components and services, as we see it, can be explained in terms of two different notions of ‘composition’, requiring two different notions of interface. In CBD, composition is integration-oriented—“the idea of component-based development is to industrialise the software development process by producing software applications by assembling prefabricated software components” [8]. In other words,

CBD addresses what, in [10] we have called ‘physiological complexity’ — the ability to build a complex system by *integrating* a number of independently developed parts. Hence, interfaces for component-based design must describe the means through which software elements can be *plugged* together to build a product and the assumptions made by each element on the environment in which it will be deployed. Interfaces in the sense of [7] – such as assume/guarantee interfaces – fall into this category: they specify the combinations of input values that components implementing an interface must accept from their environment (*assumptions*) and the combinations of output values that the environment can expect from them (*guarantees*).

In contrast, services respond to the necessity for separating “need from the need-fulfilment mechanism” [8] and address what in [10] we have called ‘social complexity’: the ability of software elements to engage with other parties to pursue a given business goal. For example, we can design a seller application that may need to use an external supplier service if the local stock is low (*the need*); the discovery and selection of, and binding to, a specific supplier (*the need-fulfilment mechanism*) are not part of the design of the seller but performed, at run time, by the underlying middleware (service-oriented architecture) according to quality-of-service constraints. In this context, service interfaces must describe the properties that are provided (so that services can be discovered) as well as those that may be required from external services (so that the middleware can select a proper provider). The latter are not assumptions on the environment as in CBD — in a sense, a service *creates* the environment that it needs to deliver what it promises.

In the context of modelling and specifying services, one can find two different kinds of approaches — choreography and orchestration — which are also reflected in the languages and standards that have been proposed for Web services, namely WS-CDL for choreography and WS-BPEL for orchestration. In a nutshell, choreography is concerned with the specification and realizability of a ‘conversation’ among a (fixed) number of peers that communicate with each other to deliver a service, whereas orchestration is concerned with the definition of a (possibly distributed) business process (or workflow) that may use external services discovered and bound to the process at run time in order to deliver a service.

Whereas the majority of formal frameworks that have been developed for SOC address choreography (see [20] for an overview), the approach that we take in this paper is orchestration-oriented. More precisely, we propose to model the workflow through which a service is orchestrated as being executed by a network of processes that interact asynchronously and offer interaction-points to which clients and external services (executed by their own networks) can bind. Hence, the questions that we propose to answer are *What is a suitable notion of interface for such asynchronous networks of processes that deliver a service?*, and *What notion of interface composition is suitable for the loose coupling of the business processes that orchestrate the interfaces?*

The rest of this paper is technical and formal. In Section 2, we present a ‘component algebra’ that is a variation on relational nets [7] adapted to the Service Component Architecture [17]. This leads us to the characterisation of services as an ‘interface algebra’ (again in the sense of [7]), which we develop in Section 3. In Section 4, we compare our framework with formal models that have been proposed in the last few years for orchestration, namely [1,3,13].

2 A Service Component Algebra

As already mentioned, we adopt the view that services are delivered by systems of components as in SCA [17]:

“SCA provides the means to compose assets, which have been implemented using a variety of technologies using SOA. The SCA composition becomes a service, which can be accessed and reused in a uniform manner. In addition, the composite service itself can be composed with other services [...] SCA service components can be built with a variety of technologies such as EJBs, Spring beans and CORBA components, and with programming languages including Java, PHP and C++ [...] SCA components can also be connected by a variety of bindings such as WSDL/SOAP web services, JavaTM Message Service (JMS) for message-oriented middleware systems and J2EETM Connector Architecture (JCA)”.

In the terminology of [7], we can see components in the sense of SCA as implementing *processes* that are connected by *channels*. However, there is a major difference in the way processes are connected. In [7], and indeed many models used for service choreography and orchestration (e.g., [1,6,18]), communication is synchronous (based in I/O connections). In order to capture the forms of loose coupling that SOAs support, communication should be asynchronous: in most business scenarios, the traditional synchronous call-and-return style of interaction is simply not appropriate. This leads us to propose a model that is closer to communicating finite-state machines [4] (also adopted in [2]) than, say, I/O automata [15]. We call our (service) component algebra *asynchronous relational nets* (ARNs) to be consistent with [7].

In an asynchronous communication model, interactions are based on the exchange of messages that are transmitted through channels (wires in the terminology of SCA). For simplicity, we ignore the data that messages may carry. We organise messages in sets that we call *ports*. More specifically, every process consists of a (finite) collection of mutually disjoint ports, i.e. each message that a process can exchange belongs to exactly one of its ports. Ports are communication abstractions that are convenient for organising networks of processes as formalised below.

Every message belonging to a port has an associated *polarity*: $-$ if it is an outgoing message (published at the port) and $+$ if it is incoming (delivered at the port). This is the notation proposed in [4] and also adopted in [1].

Definition 1 (Ports and message polarity). A port is a set of messages. Every port M has a partition $M^- \cup M^+$. The messages in M^- are said to have polarity $-$, and those in M^+ have polarity $+$.

The actions of sending (publishing) or receiving (being delivered) a message m are denoted by $m!$ and m_i , respectively. In the literature, one typically finds $m?$ for the latter. In our model, we use $m?$ for the action of processing the message and m_i for the action of discarding the message: processes should not refuse the delivery of messages but they should be able to discard them.

Definition 2 (Actions). Let M be a port and $m \in M$.

- If $m \in M^-$, the set of actions associated with m is $A_m = \{m!\}$ and, if $m \in M^+$, $A_m = \{m_i, m?, m_\zeta\}$
- The set of actions associated with M is $A_M = \bigcup_{m \in M} A_m$.

In [7], predicates are used as a means of describing properties of input/output behaviour, i.e. establishing relations (or the lack thereof) between inputs and outputs of processes, leading to several classes of relational nets depending on when they are considered to be ‘well-formed’. In the context of our asynchronous communication model, behaviour is observed in terms of the actions that are performed, for which the natural formalism to use is temporal logic (e.g., [14]). For simplicity, we use linear temporal logic, i.e. we observe traces of actions. In order not to constrain the environment in which processes execute and communicate, we take traces to be infinite and we allow several actions to occur ‘simultaneously’, i.e. the granularity of observations may not be so fine that we can always tell which of two actions occurred first.

Definition 3 (LTL). Let A be a set (of actions). We use a classical linear temporal logic where every $a \in A$ is an atomic formula and formulas are interpreted over infinite traces $\lambda \in (2^A)^\omega$. For every collection Φ of formulas, we define:

- $\Lambda_\Phi = \{\lambda \in (2^A)^\omega : \forall \phi \in \Phi (\lambda \models \phi)\}$
- $\Pi_\Phi = \{\pi \in (2^A)^*: \exists \lambda \in \Lambda_\Phi (\pi \prec \lambda)\}$ where $\pi \prec \lambda$ means that π is a prefix of λ . Given $\pi \in (2^A)^*$ and $B \subseteq A$, we denote by $(\pi \cdot B)$ the trace obtained by extending π with B .

We say that a collection Φ of formulas entails ϕ — $\Phi \models \phi$ — iff $\Lambda_\Phi \subseteq \Lambda_\phi$. We say that a collection Φ of formulas is consistent iff $\Lambda_\Phi \neq \emptyset$.

The fact that, at any given point i , it is possible that $\lambda(i)$ is empty means that we are using an open semantics, i.e. we are considering transitions during which the ARN is idle. This means that we can use the logic to reason about global properties of networks of processes, which is convenient for giving semantics to the composition of ARNs. Infinite traces are important because, even if the execution of individual processes in a service session is, in typical business applications, finite, the ARN may bind to other ARNs at run time as a result of the discovery of required services. Unbounded behaviour may indeed arise in SOC because of the intrinsic dynamics of the configurations that execute business applications, i.e. it is the configuration that is unbounded, not the behaviour of the processes and channels that execute in the configuration.

In this paper, we work with descriptions (sets of formulas) over different sets of actions, which requires that we are able to map between the corresponding languages:

Proposition and Definition 4 (Translation). Let $\sigma: A \rightarrow B$ be a function. Given an LTL formula ϕ over A , we define its translation $\sigma(\phi)$ as the formula over B that is obtained by replacing every action $a \in A$ by $\sigma(a)$. The following properties hold:

- For every $\lambda \in 2^{B^\omega}$, $\lambda \models \sigma(\phi)$ iff $\sigma^{-1}(\lambda) \models \phi$ where $\sigma^{-1}(\lambda)(i)$ is $\sigma^{-1}(\lambda(i))$.
- Any set Φ of LTL formulas over A is consistent if $\sigma(\Phi)$ is consistent.
- For every set Φ of LTL formulas over A and formula ψ also over A , if $\Phi \models \psi$ then $\sigma(\Phi) \models \sigma(\psi)$.

Furthermore, if σ is an injection, the implications above are equivalences.

Proof. The first property is easily proved by structural induction, from which the other two follow. The properties of injections are proved in the same way on the direct image.

Notice that, in the case of injections, the translations induce conservative extensions, i.e. $\sigma(\Phi)$ is a conservative translation of Φ . We are particularly interested in translations that, given a set A and a symbol p , prefix the elements of A with ‘ $p.$ ’. We denote these translations by $(p.-)$. Note that prefixing defines a bijection between A and its image.

Definition 5 (Process). A process consists of:

- A finite set γ of mutually disjoint ports.
- A consistent set Φ of LTL formulas over $\bigcup_{M \in \gamma} A_M$.

Fig. 1 presents an example of a process *Seller* with two ports. In the port depicted on the left, which we designate by L_{sl} , it receives the message *buy* and sends messages *price* and *fwd_details*. The other port, depicted on the right and called R_{sl} , has incoming message *details* and outgoing message *product*. Among other properties, we can see that *Seller* ensures to eventually sending the messages *product* and *price* in reaction to the delivery of *buy*. As explained below, the grouping of messages in ports implies that, whilst *price* is sent over the channel that transmits *buy*, *product* is sent over a different channel.

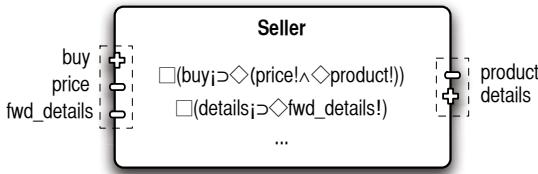


Fig. 1. Example of a process with two ports

Interactions in ARNs are established through channels. Channels transmit messages both ways, i.e. they are bidirectional, which is consistent with [4]. Notice that, in some formalisms (e.g., [2]), channels are unidirectional, which is not so convenient for capturing typical forms of conversation that, like in SCA, are two-way: a request sent by the sender through a wire has a reply sent by the receiver through the same wire (channel). This means that channels are agnostic in what concerns the polarity of messages: these are only meaningful within ports.

Definition 6 (Channel). A channel consists of:

- A set M of messages.
- A consistent set Φ of LTL formulas over $A_M = \{m!, m_j : m \in M\}$.

Notice that in [2] as well as other asynchronous communication models adopted for choreography, when sent, messages are inserted in the queue of the consumer. In the context of loose coupling that is of interest for SOC, channels (wires) may have a behaviour of their own that one may wish to describe or, in the context of interfaces, specify. Therefore, for generality, we take channels as first-class entities that are responsible for delivering messages.

Channels connect processes through ports that assign opposite polarities to messages. Formally, the connections are established through what we call attachments:

Definition 7 (Connection). Let M_1 and M_2 be ports and $\langle M, \Phi \rangle$ a channel. A connection between M_1 and M_2 via $\langle M, \Phi \rangle$ consists of a pair of bijections $\mu_i : M \rightarrow M_i$ such that $\mu_i^{-1}(M_i^+) = \mu_j^{-1}(M_j^-)$, $\{i, j\} = \{1, 2\}$. Each bijection μ_i is called the attachment of $\langle M, \Phi \rangle$ to M_i . We denote the connection by $\langle M_1 \xleftarrow{\mu_1} M \xrightarrow{\mu_2} M_2, \Phi \rangle$.

Proposition 8. Every connection $\langle M_1 \xleftarrow{\mu_1} M \xrightarrow{\mu_2} M_2, \Phi \rangle$ defines an injection $\langle \mu_1, \mu_2 \rangle$ from A_M to $A_{M_1} \cup A_{M_2}$ as follows: for every $m \in M$ and $\{i, j\} = \{1, 2\}$, if $\mu_i(m) \in M_i^-$ then $\langle \mu_1, \mu_2 \rangle(m!) = \mu_i(m)!$ and $\langle \mu_1, \mu_2 \rangle(m_j) = \mu_j(m)_j$.

Definition 9 (Asynchronous relational net). An asynchronous relational net (ARN) α consists of:

- A simple finite graph $\langle P, C \rangle$ where P is a set of nodes and C is a set of edges. Note that each edge is an unordered pair $\{p, q\}$ of nodes.
- A labelling function that assigns a process to every node and a connection to every edge such that:
 - If $p : \langle \gamma, \Phi \rangle$ and $q : \langle \gamma', \Phi' \rangle$ then $\{p, q\}$ is labelled with a connection of the form $\langle M_p \xleftarrow{\mu_p} M \xrightarrow{\mu_q} M_q, \Phi'' \rangle$ where $M_p \in \gamma$ and $M_q \in \gamma'$.
 - For every $\{p, q\} : \langle M_p \xleftarrow{\mu_p} M \xrightarrow{\mu_q} M_q, \Phi \rangle$ and $\{p, q'\} : \langle M'_p \xleftarrow{\mu'_p} M' \xrightarrow{\mu'_{q'}} M'_{q'}, \Phi' \rangle$, if $q \neq q'$ then $M_p \neq M'_p$.

We also define the following sets:

- $A_p = p.(\bigcup_{M \in \gamma_p} A_M)$ is the language associated with p ,
- $A_\alpha = \bigcup_{p \in P} A_p$ is the language associated with α ,
- $A_c = \langle p_- \circ \mu_p, q_- \circ \mu_q \rangle(A_M)$ is the language associated with $\gamma_c : \langle M_p \xleftarrow{\mu_p} M \xrightarrow{\mu_q} M_q \rangle$.
- Φ_α is the union of the following sets of formulas
 - For every $p : \langle \gamma, \Phi \rangle$, the prefix-translation Φ_p of Φ by (p_-) .
 - For every $c : \langle M_p \xleftarrow{\mu_p} M \xrightarrow{\mu_q} M_q, \Phi \rangle$, the translation $\Phi_c = \langle p_- \circ \mu_p, q_- \circ \mu_q \rangle(\Phi)$
- $\Lambda_\alpha = \{\lambda \in 2^{A_\alpha^\omega} : \forall p \in P (\lambda|_{A_p} \in \Lambda_{\Phi_p}) \wedge \forall c \in C (\lambda|_{A_c} \in \Lambda_{\Phi_c})\}$
The set of infinite traces that are projected to models of all processes and channels.
- $\Pi_\alpha = \{\pi \in 2^{A_\alpha^*} : \forall p \in P (\pi|_{A_p} \in \Pi_{\Phi_p}) \wedge \forall c \in C (\pi|_{A_c} \in \Pi_{\Phi_c})\}$
The set of finite traces that are projected to prefixes of models of all processes and channels.

We often refer to the ARN through the quadruple $\langle P, C, \gamma, \Phi \rangle$ where γ returns the set of ports of the processes that label the nodes and the pair of attachments of the connections that label the edges, and Φ returns the corresponding descriptions. The fact that the graph is simple — undirected, without self-loops or multiple edges — means that all interactions between two given processes are supported by a single channel and that no process can interact with itself. The graph is undirected because, as already mentioned, channels are bidirectional. Furthermore, different channels cannot share ports.

Notice that nodes and edges denote *instances* of processes and channels, respectively. Different nodes (resp. edges) can be labelled with the same process (resp. channel). Therefore, in order to reason about the properties of the ARN as a whole we need to

translate the descriptions of the processes and channels involved to a language in which we can distinguish between the corresponding instances. The set Φ_α consists precisely of the translations of all the descriptions of the processes and channels using the nodes as prefixes for the actions that they execute. Notice that, by Prop. 4, these translations are conservative, i.e. neither processes nor channels gain additional properties because of the translations. However, by taking the union of all such descriptions, new properties may emerge, i.e. Φ_α is not necessarily a conservative extension of the individual descriptions.

A process in isolation, such as *Seller* (see Fig. 1), defines an ARN. Fig. 2 presents another ARN that also involves *Seller*. In this ARN, the port R_{sl} of *Seller* is connected with the port M_{sp} of process *Supplier*, which consists of the incoming message *request* and the outgoing message *invoice*. The channel that connects R_{sl} and M_{sp} is described to be reliable with respect to *product*: it ensures to delivering *product*, which *Supplier* receives under the name *request*. Formally, this ARN consists of a graph with two nodes $sl:\text{Seller}$ and $sp:\text{Supplier}$ and one edge $\{sl, sp\}:w_{ss}$, where w_{ss} is the connection

$$\langle R_{sl} \xleftarrow{\mu_{sl}} \{m, n\} \xrightarrow{\mu_{sp}} M_{sp}, \{\square(m! \supset \diamond m_i)\} \rangle$$

with $\mu_{sl} = \{m \mapsto \text{product}, n \mapsto \text{details}\}$, $\mu_{sp} = \{m \mapsto \text{request}, n \mapsto \text{invoice}\}$.

The set $\Phi_{\text{SELLERWITHSUPPLIER}}$ consists of the translation of all properties of its processes and connections. Hence, it includes:

- $\square(sl.\text{buy}_i \supset \diamond(sl.\text{price}! \wedge \diamond sl.\text{product}!))$
- $\square(sl.\text{details}_i \supset \diamond sl.\text{fwd_details}!)$
- $\square(sp.\text{request}_i \supset \diamond sp.\text{invoice}!)$
- $\square(sl.\text{product}! \supset \diamond sp.\text{request}_i)$

Notice that the last formula, which is the translation of the description of the channel, relates the languages of *Seller* and *Supplier*: the publication of *product* by *Seller* leads to the delivery of *request* to *Supplier*. In this context, *product* and *request* are just local names of the ‘same’ message as perceived by the two processes being connected. The ability to operate with local names is essential for SOC because, in the context of run-time discovery and binding, it is not possible to rely on a shared name space. This is why it is important that channels are first-class entities, i.e., that communication is established explicitly by correlating the actions that represent the local view that each party has of a message exchange.

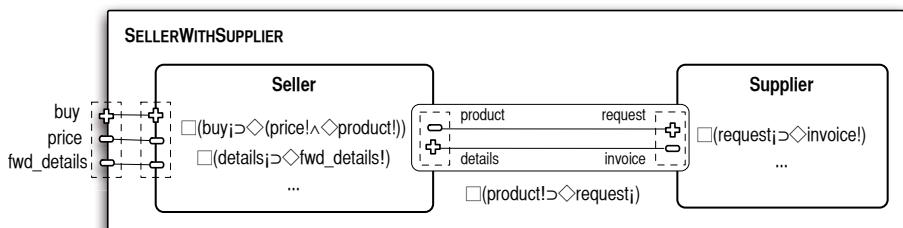


Fig. 2. An example of an ARN with two processes connected through a channel

In [7], joint consistency of the descriptions of the processes and the connections, i.e., of Φ_α , would be required for the ARN to be well defined. However, consistency does not ensure that the processes will always be able to make progress while interacting through the channels, which is why we prefer instead to use the following property as a criterion for well-formedness:

Definition 10 (Progress-enabled ARN). *We say that an ARN α is progress-enabled iff $\forall \pi \in \Pi_\alpha \exists A \subseteq A_\alpha (\pi \cdot A) \in \Pi_\alpha$.*

It is not difficult to see that any ARN α with a single process, such as *Seller*, is progress-enabled. This is because the process is isolated. In general, not every port of every process is necessarily connected to a port of another process. Such ports provide the points through which the ARN can interact with other ARNs. For example, *SELLERWITHSUPPLIER* has a single interaction point, which in Fig. 2 is represented by projecting the corresponding port to the external box.

Definition 11 (Interaction-point). *An interaction-point of an ARN $\alpha = \langle P, C, \gamma, \Phi \rangle$ is a pair $\langle p, M \rangle$ such that $p \in P$, $M \in \gamma_p$ and there is no edge $\{p, q\} \in C$ labelled with a connection that involves M . We denote by I_α the collection of interaction-points of α .*

Interaction-points are used in the notion of composition that we define for ARNs, which also subsumes the notion of interconnect of [7]:

Proposition and Definition 12 (Composition of ARNs). *Let $\alpha_1 = \langle P_1, C_1, \gamma_1, \Phi_1 \rangle$ and $\alpha_2 = \langle P_2, C_2, \gamma_2, \Phi_2 \rangle$ be ARNs such that P_1 and P_2 are disjoint, and a family $w^i = \langle M_1^i \xleftarrow{\mu_1^i} M \xrightarrow{\mu_2^i} M_2^i, \Psi^i \rangle$ ($i = 1 \dots n$) of connections for interaction-points $\langle p_1^i, M_1^i \rangle$ of α_1 and $\langle p_2^i, M_2^i \rangle$ of α_2 such that $p_1^i \neq p_2^j$ if $i \neq j$ and $p_2^i \neq p_2^j$ if $i \neq j$. The composition*

$$\alpha_1 \parallel_{\langle p_1^i, M_1^i \rangle, w^i, \langle p_2^i, M_2^i \rangle}^{i=1\dots n} \alpha_2$$

is the ARN defined as follows:

- Its graph is $\langle P_1 \cup P_2, C_1 \cup C_2 \cup \bigcup_{i=1\dots n} \{p_1^i, p_2^i\} \rangle$
- Its labelling function coincides with that of α_1 and α_2 on the corresponding subgraphs, and assigns to the new edges $\{p_1^i, p_2^i\}$ the label w^i .

Proof. We need to prove that the composition does define an ARN. This is because we are adding to the sum of the graphs edges between interaction-points that do not share interaction-points, the resulting graph is simple. It is easy to check that the labels are well defined.

Fig. 2 can also be used to illustrate the composition of ARNs: *SELLERWITHSUPPLIER* is the composition of the two single-process ARNs defined by *Seller* and *Supplier* via the connection w_{ss} .

Given that we are interested in ARNs that are progress-enabled, it would be useful to have criteria for determining when a composition of progress-enabled ARNs is still progress-enabled. For this purpose, an important property of an ARN relative to its set of interaction-points is that it does not constrain the actions that do not ‘belong’

to the ARN. Naturally, this needs to be understood in terms of a computational and communication model in which it is clear what dependencies exist between the different parties. As already mentioned, we take it to be the responsibility of processes to publish and process messages, and of channels to deliver them. This requires that processes are able to buffer incoming messages, i.e., to be ‘delivery-enabled’, and that channels are able to buffer published messages, i.e., to be ‘publication-enabled’.

Definition 13 (Delivery-enabled). Let $\alpha = \langle P, C, \gamma, \Phi \rangle$ be an ARN, $\langle p, M \rangle \in I_\alpha$ one of its interaction-points, and $D_{\langle p, M \rangle} = \{p.m_j : m \in M^+\}$. We say that α is delivery-enabled in relation to $\langle p, M \rangle$ if, for every $(\pi \cdot A) \in \Pi_\alpha$ and $B \subseteq D_{\langle p, M \rangle}$, $(\pi \cdot B \cup (A \setminus D_{\langle p, M \rangle})) \in \Pi_\alpha$.

The property requires that any prefix can be extended by any set of messages delivered at one of its interaction-points. Considering again the ARN defined by *Seller*, if its description is limited to the formulas shown in Fig. 1, then it is not difficult to conclude that the ARN is delivery-enabled for both its interaction-points — the constraints put on the delivery of *buy* and *details* are both satisfiable.

Definition 14 (Publication-enabled). Let $h = \langle M, \Phi \rangle$ be a channel and $E_h = \{m! : m \in M\}$. We say that h is publication-enabled iff, for every $(\pi \cdot A) \in \Pi_\Phi$ and $B \subseteq E_h$, we have $\pi \cdot (B \cup (A \setminus E_h)) \in \Pi_\Phi$.

The requirement here is that any prefix can be extended by the publication of a set of messages, i.e. the channel should not prevent processes from publishing messages. For example, the channel used in the connection w_{ss} is clearly publication-enabled: the extension of any prefix $(\pi \cdot A)$ in $\Pi_{\Phi_{w_{ss}}}$ with the publication of m (if it was not already in A) is still a prefix of an infinite trace that satisfies $\square(m! \supset \diamond m_1)$ by executing m_1 at a later stage.

Theorem 15. Let $\alpha = (\alpha_1 \parallel_{\langle p_1^i, M_1^i \rangle, w^i, \langle p_2^i, M_2^i \rangle}^{i=1\dots n} \alpha_2)$ be a composition of progress-enabled ARNs where, for each $i = 1 \dots n$, $w^i = \langle M_1^i \xrightarrow{\mu_1^i} M \xrightarrow{\mu_2^i} M_2^i, \Psi^i \rangle$. If, for each $i = 1 \dots n$, α_1 is delivery-enabled in relation to $\langle p_1^i, M_1^i \rangle$, α_2 is delivery-enabled in relation to $\langle p_2^i, M_2^i \rangle$ and $h^i = \langle M^i, \Phi^i \rangle$ is publication-enabled, then α is progress-enabled.

We can use this theorem to prove that the ARN presented in Fig. 2 is progress-enabled. Because, as already argued, the channel used in this composition is publication-enabled and *Seller* is delivery-enabled, it would remain to prove that so is *Supplier*, which is similar to the case of *Seller*.

Proposition 16. Let $\alpha = (\alpha_1 \parallel_{\langle p_1^i, M_1^i \rangle, w^i, \langle p_2^i, M_2^i \rangle}^{i=1\dots n} \alpha_2)$ be a composition.

- Let $\langle p'_1, M'_1 \rangle$ be an interaction-point of α_1 different from all $\langle p_1^i, M_1^i \rangle$. If α_1 is delivery-enabled in relation to $\langle p'_1, M'_1 \rangle$, so is α .
- Let $\langle p'_2, M'_2 \rangle$ be an interaction-point of α_2 different from all $\langle p_2^i, M_2^i \rangle$. If α_2 is delivery-enabled in relation to $\langle p'_2, M'_2 \rangle$, so is α .

3 A Service Interface Algebra

In this section, we put forward a notion of interface for software components described in terms of ARNs and a notion of interface composition that is suitable for service-oriented design. As discussed in Section 1, this means that interfaces need to specify the services that customers can expect from ARNs as well as the dependencies that the ARNs may have on external services for providing the services that they offer.

In our model, a service interface identifies a number of ports through which services are provided and ports through which services are required (hence the importance of ports for correlating messages that belong together from a business point of view). Temporal formulae are used for specifying the properties offered or required.

Ports for required services include messages as sent or received by the external service. Therefore, to complete the interface we need to be able to express requirements on the channel through which communication with the external service will take place, if and when required. In order to express those properties, we need to have actions on both sides of the channel, for which we introduce the notion of dual port.

Definition 17 (Dual port). Given a port M , we denote by M^{op} the port defined by $M^{op+} = M^-$ and $M^{op-} = M^+$.

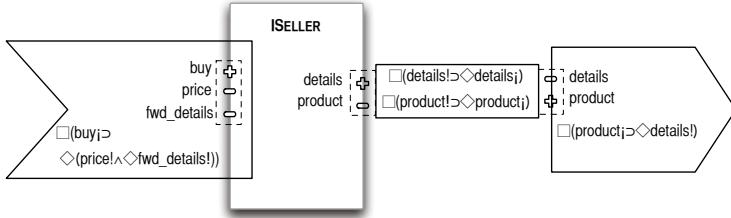
Notice that $(M^{op})^{op} = M^{op}$.

Definition 18 (Service interface). A service interface i consists of:

- A set I (of interface-points) partitioned into two sets I^\rightarrow and I^\leftarrow the members of which are called the provides- and requires-points, respectively.
- For every interface-point r , a port M_r .
- For every point $r \in I^\rightarrow$, a consistent set of LTL formulas Φ_r over A_{M_r} .
- For every point $r \in I^\leftarrow$:
 - a consistent set of LTL formulas Φ_r over A_{M_r} making $\alpha_r = \langle \{r\}, \emptyset, M_r, \Phi_r \rangle$ delivery-enabled (see Def. 13),
 - a consistent set of LTL formulas Ψ_{M_r} over $\{m!, m; : m \in M_r\}$ making $\langle M_r, \Psi_{M_r} \rangle$ a publication-enabled channel (see Def. 14).

We identify an interface with the tuple $\langle I^\rightarrow, I^\leftarrow, M, \Phi, \Psi \rangle$ where $M_r : r \in I$, $\Phi_r : r \in I^\rightarrow$, $\Psi_r : r \in I^\leftarrow$ are the indexed families that identify the ports and specifications of each point of the interface. Notice that different points may have the same port, i.e., ports are types.

The sets of formulas at each interface-point specify the protocols that services require from external services (in the case of requires-points) and offer to clients (in the case of provides-ports). For example, Fig. 3 presents a service interface with one provides and one requires-point (we use a graphical notation similar to that of SCA). On the left, we have an interaction point p through which the service is provided and, on the right side, the interaction point r through which an external service is required. According to what is specified, the service offers, in reaction to the delivery of the message *buy*, to reply by publishing the message *price* followed eventually by *fwd_details*. On the other hand, the required service is asked to react to the delivery of *product* by publishing *details*.

**Fig. 3.** An example of a service interface

The connection with the external service is required to ensure that the transmission of both messages is reliable.

Notice that the properties specified of the interface-points play a role that is different from the assumption/guarantee (A/G) specifications that have been proposed (since [16]) for networks of processes and also used in [19] for web services. The aim of A/G is to ensure compositionality of specifications of processes by making explicit assumptions about the way they interact with their environment. The purpose of the interfaces that we propose is, instead, to specify the protocols offered to clients of the service and the protocols that the external services that the service may need to discover and bind to are required to follow. This becomes clear in the definition of the notion of implementation of a service interface, which we call an orchestration. Compositionality is then proved (Theo. 22) under the assumptions made on the requires-points, namely delivery and publication enabledness, which concern precisely the way processes and channels interfere with their environments. That is, our interfaces do address the interference between service execution and their environment, but the formulas associated with requires-points are not assumptions and those of provides-ports are not guarantees in the traditional sense of A/G specifications.

Definition 19 (Orchestration). An orchestration of a service interface $\langle I^\rightarrow, I^\leftarrow, M, \Phi, \Psi \rangle$ consists of:

- An ARN $\alpha = \langle P, C, \gamma, \Phi \rangle$ where P and I are disjoint, which is progress-enabled and delivery-enabled in relation to all its interaction-points.
- A one-to-one correspondence ρ between I and I_α ; we will write $r \xrightarrow{\rho} p$ to indicate that $\rho(r) = \langle p, M_p \rangle$ for some port M_p .
- For every $r \in I^\rightarrow$, a polarity-preserving bijection $\rho_r : M_r \rightarrow M_p$ where $r \xrightarrow{\rho} p$.
- For every $r \in I^\leftarrow$, a polarity-preserving bijection $\rho_r : M_r^{\text{op}} \rightarrow M_p$ where $r \xrightarrow{\rho} p$.

Let $\alpha^* = (\alpha \parallel_{\rho(r), w_r, \langle r, M_r \rangle} \alpha_r)_{r \in I^\leftarrow}$ with $w_r = \langle M_p \xleftarrow{\rho_r} M_r \xrightarrow{\text{id}} M_r, \Psi_r \rangle$, $r \xrightarrow{\rho} p$. We require that, for every $r \in I^\rightarrow$, $(p \multimap \rho_r)^{-1}(\Lambda_{\alpha^*}) \subseteq \Lambda_{\Phi_r}$ —equivalently, $\Phi_{\alpha^*} \models p.(\rho_r(\Phi_r))$. A service interface that can be orchestrated is said to be consistent.

The condition requires that every model of the ARN composed with the requires-points and channels be also a model of the specifications of the provides-points. That is, no matter what the external services that bind to the requires-points do and how the channels transmit messages (as long as they satisfy the corresponding specifications), the

ARN will be able to operate and deliver the properties specified in the provides-points. Notice that, by Theo. 15 the composition is progress-enabled.

Also note that provides-points are mapped to interaction-points of the ARN preserving the polarity of the messages, but requires-points reverse the polarity. This is because every requires-point $r \in I^{\leftarrow}$ represents the external service that is required whereas $r \xrightarrow{\rho} p$ identifies the interaction-point through which that external service, once discovered, will bind to the orchestration. The ARNs α_r represent those external services.

Consider again the ARN defined by *Seller* as in Fig. 1. It is not difficult to see that, together with the correspondences $p \mapsto \langle \text{Seller}, L_{sl} \rangle$ and $r \mapsto \langle \text{Seller}, R_{sl} \rangle$, *Seller* defines an orchestration for the service interface ISSELLER. Indeed, according to the definition above, Seller^* is the composition

$$\text{Seller} \quad || \quad \langle \{r\}, \emptyset, R_{sl}^{op}, \{\Box(\text{product}_i \supset \Diamond \text{details}_i)\} \rangle \\ \langle sl, R_{sl} \rangle, w_r, \langle r, R_{sl}^{op} \rangle$$

Hence, the set Φ_{Seller^*} includes the following properties:

1. $\Box(sl.\text{buy}_i \supset \Diamond(sl.\text{price}_i \wedge \Diamond sl.\text{product}_i))$ (from *Seller*)
2. $\Box(sl.\text{details}_i \supset \Diamond sl.\text{fwd_details}_i)$ (from *Seller*)
3. $\Box(r.\text{product}_i \supset \Diamond r.\text{details}_i)$ (from the specification Φ_r of the requires-point)
4. $\Box(r.\text{details}_i \supset \Diamond sl.\text{details}_i)$ (from the specification Ψ_r of the required channel)
5. $\Box(sl.\text{product}_i \supset \Diamond r.\text{product}_i)$ (from Ψ_r)

It is not difficult to conclude that $\Box(p.\text{buy}_i \supset \Diamond(sl.\text{price}_i \wedge \Diamond sl.\text{fwd_details}_i))$ is a logical consequence of Φ_{Seller^*} . We just have to produce a chain of implications using (1), (5), (3), (4) and (2), in this order.

Definition 20 (Match). A match between two interfaces $i = \langle I^{\rightarrow}, I^{\leftarrow}, M^i, \Phi^i, \Psi^i \rangle$ and $j = \langle J^{\rightarrow}, J^{\leftarrow}, M^j, \Phi^j, \Psi^j \rangle$ is a family of triples $\langle r^m, s^m, \delta^m \rangle$, $m = 1 \dots n$, where $r^m \in I^{\leftarrow}$, $s^m \in J^{\rightarrow}$ and $\delta^m: M_{r^m}^i \rightarrow M_{s^m}^j$ is a polarity-preserving bijection such that $\Phi_{s^m}^j \models \delta^m(\Phi_{r^m}^i)$. Two interfaces are said to be compatible if their sets of interface-points are disjoint and admit a match.

That is, a match maps certain requires-points of one of the interfaces to provides-points of the other in such a way that the required properties are entailed by the provided ones. Notice that, because the identity of the interface-points is immaterial, requiring that the sets of points of the interfaces be disjoint is not restrictive at all. We typically use $\delta^m: r^m \rightarrow s^m$ to refer to a match.

Definition 21 (Composition of interfaces). Given a match $\delta^m: r^m \rightarrow s^m$ between compatible interfaces $i = \langle I^{\rightarrow}, I^{\leftarrow}, M^i, \Phi^i, \Psi^i \rangle$ and $j = \langle J^{\rightarrow}, J^{\leftarrow}, M^j, \Phi^j, \Psi^j \rangle$, their composition $(i \parallel_{\delta^m: r^m \rightarrow s^m} j) = \langle K^{\rightarrow}, K^{\leftarrow}, M, \Phi, \Psi \rangle$ is defined as follows:

- $K^{\rightarrow} = I^{\rightarrow} \cup (J^{\rightarrow} \setminus \{s^m : m = 1 \dots n\})$.
- $K^{\leftarrow} = J^{\leftarrow} \cup (I^{\leftarrow} \setminus \{r^m : m = 1 \dots n\})$.
- $\langle M, \Phi, \Psi \rangle$ coincides with $\langle M^i, \Phi^i, \Psi^i \rangle$ and $\langle M^j, \Phi^j, \Psi^j \rangle$ on the corresponding points.

Notice that the composition of interfaces is not commutative: one of the interfaces plays the role of client and the other of supplier of services.

We can now prove compositionality, i.e., that the composition of the orchestrations of compatible interfaces is an orchestration of the composition of the interfaces.

Theorem 22 (Composition of orchestrations). *Let $i = \langle I^\rightarrow, I^\leftarrow, M^i, \Phi^i, \Psi^i \rangle$ and $j = \langle J^\rightarrow, J^\leftarrow, M^j, \Phi^j, \Psi^j \rangle$ be compatible interfaces, $\delta^m : r^m \rightarrow s^m$ a match between them, and $\langle \alpha, \rho \rangle$ and $\langle \beta, \sigma \rangle$ orchestrations of i and j , respectively, with disjoint graphs.*

$$(\alpha \parallel_{\langle p^m, M_{p^m} \rangle, w^m, \langle q^m, M_{q^m} \rangle}^{m=1..n} \beta)$$

where $w^m = \langle M_{p^m} \xleftarrow{\rho_{r^m}} M_{r^m} \xrightarrow{\sigma_{s^m} \circ \delta^m} M_{q^m}, \Psi_{M_{r^m}}^i \rangle$, $\rho(r^m) = \langle p^m, M_{p^m} \rangle$ and $\sigma(s^m) = \langle q^m, M_{q^m} \rangle$ defines an orchestration of $(i \parallel_{\delta^m : r^m \rightarrow s^m} j)$ through the mapping κ that coincides with ρ on I and with σ on J .

Compositionality is one of the key properties required in [7] for a suitable notion of interface. From the software engineering point of view, it means that there is indeed a separation between interfaces and their implementations in the sense that, at the design level, composition can be performed at the interface level independently of the way the interfaces will be implemented. In particular, one can guarantee that the composition of compatible interfaces can indeed be orchestrated, which is captured by the following corollary (the theorem provides a concrete way of deriving that orchestration from those of the component interfaces):

Corollary 23 (Preservation of consistency). *The composition of two compatible and consistent interfaces is consistent.*

4 Related Work and Concluding Remarks

In this paper, we took inspiration from the work reported in [7] on a theory of interfaces for component-based design to propose a formalisation of ‘services’ as interfaces for an algebra of asynchronous components. That is, we exposed and provided mathematical support for the view that services are, at a certain level of abstraction, a way of *using* software components — what is sometimes called a ‘service-overlay’ — and not so much a way of *constructing* software, which is consistent with the way services are being perceived in businesses [8] and supported by architectures such as SCA [17].

This view differs from the more traditional component-based approach in which components expose methods in their interfaces and bind tightly to each other (based on I/O-relations) to construct software applications. In our approach, components expose conversational, stateful interfaces through which they can discover and bind, on the fly, to external services or expose services that can be discovered by business applications. Having in mind that one of the essential features of SOC is loose-binding, we proposed a component algebra that is asynchronous — essentially, an asynchronous version of relational nets as defined in [7].

As mentioned in Section 1, most formal frameworks that have been proposed for SOC address *choreography*, i.e., the specification of a global conversation among a

fixed number of peers and the way it can be realised in terms of the local behaviour generated by implementations of the peers. A summary of different choreography models that have been proposed in the literature can be found in [20]. Among those, we would like to distinguish the class of automata-based models proposed in [2,5,13], which are asynchronous. Such choreography models are inherently different from ours in the sense that they study different problems: the adoption of automata reflects the need to study the properties and realisability of conversation protocols captured as words of a language of message exchange. It would be tempting to draw a parallel between their notion of composite service — a network of machines — and our ARNs, but they are actually poles apart: our aim has not been to model the conversations that characterise the global behaviour of the peers that deliver a service, but to model the network of processes executed by an individual peer and how that network orchestrates a service interface for that peer — that is, our approach is *orchestration-based*. Therefore, we do not make direct usage of automata, although a reification of our processes could naturally be given in terms of automata. Our usage of temporal logic for describing ARNs, as a counterpart to the use of first-order logic in [7] for describing I/O communication, has the advantage of being more abstract than a specific choice of an automata-based model (or, for that matter, a Petri-net model [18]). This has also allowed us to adopt a more general model of asynchronous communication in which channels are first-class entities (reflecting the importance that they have in SOC). We are currently studying decidability and other structural properties of our model and the extent to which we can use model-checking or other techniques to support analysis.

Another notion of web service interface has been proposed in [3]. This work presents a specific language, not a general approach like we did in this paper, but there are some fundamental differences between them, for example in the fact that their underlying model of interaction is synchronous (method invocation), which is not suitable for loose coupling. The underlying approach is, like ours, orchestration-based but, once again, more specific than ours in that orchestrations are modelled through a specific class of automata supporting a restricted language of temporal logic specifications. Another fundamental difference is that, whereas in [3] the orchestration of a service is provided by an automaton, ours is provided by a network of processes (as in SCA), which provides a better model for capturing the dynamic aspects of SOC that arise from run-time discovery and binding: our notion of composition is not for integration (as in CBD) but for dynamic interconnection of processes. This is also reflected in the notion of interface: the interfaces used in [3] are meant for design-time composition, the client being statically bound to the invoked service (which is the same for all invocations); the interfaces that we proposed address a different form of composition in which the provider (the “need-fulfilment mechanism”) is procured at run time and, therefore, can differ from one invocation to the next, as formalised in [11] in a more general algebraic setting.

Being based on a specific language, [3] explores a number of important issues related to compatibility and consistency that arise naturally in service design when one considers semantically-rich interactions, e.g., when messages carry data or are correlated according to given business protocols. A similar orchestration-based approach has been presented in [1], which is also synchronous and based on finite-state machines, and also addresses notions of compatibility and composition of conversation protocols (though,

interestingly, based on branching time). We are studying an extension of our framework that can support such richer models of interaction (and the compatibility issues that they raise), for which we are using, as a starting point, the model that we adopted in the language SRML [12], which has the advantage of being asynchronous.

Although we consider that the main contribution of this paper is to put forward a notion of interface that can bring service-oriented design to the ‘standards’ of component-based design, there are still aspects of the theory of component interfaces developed in [7] that need to be transposed to services. For example, an important ingredient of that theory is a notion of compositional refinement that applies to interfaces (for top-down design) and a notion of compositional abstraction for implementations (orchestrations in the case of services), that can support bottom-up verification.

Other lines for further work concern extensions to deal with time, which is critical for service-level agreements, and to address the run-time discovery, selection and binding processes that are intrinsic to SOC. We plan to use, as a starting point, the algebraic semantics that we developed for SRML [11]. Important challenges that arise here relate to the unbounded nature of the configurations (ARNs) that execute business applications in a service-oriented setting, which is quite different from the complexity of the processes and communication channels that execute in those configurations.

Acknowledgments

We would like to thank Nir Piterman and Emilio Tuosto for many helpful comments and suggestions.

References

1. Benatallah, B., Casati, F., Toumani, F.: Representing, analysing and managing web service protocols. *Data Knowl. Eng.* 58(3), 327–357 (2006)
2. Betin-Can, A., Bultan, T., Fu, X.: Design for verification for asynchronously communicating web services. In: Ellis and Hagino [9], pp. 750–759
3. Beyer, D., Chakrabarti, A., Henzinger, T.A.: Web service interfaces. In: Ellis and Hagino [9], pp. 148–159
4. Brand, D., Zafiropulo, P.: On communicating finite-state machines. *J. ACM* 30(2), 323–342 (1983)
5. Bultan, T., Fu, X., Hull, R., Su, J.: Conversation specification: a new approach to design and analysis of e-service composition. In: WWW, pp. 403–410 (2003)
6. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: De Nicola, R. (ed.) *ESOP 2007. LNCS*, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)
7. de Alfaro, L., Henzinger, T.A.: Interface theories for component-based design. In: Henzinger, T.A., Kirsch, C.M. (eds.) *EMSOFT 2001. LNCS*, vol. 2211, pp. 148–165. Springer, Heidelberg (2001)
8. Elfatatty, A.: Dealing with change: components versus services. *Commun. ACM* 50(8), 35–39 (2007)
9. Ellis, A., Hagino, T. (eds.): *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14. ACM, New York* (2005)

10. Fiadeiro, J.L.: Designing for software's social complexity. *IEEE Computer* 40(1), 34–39 (2007)
11. Fiadeiro, J.L., Lopes, A., Bocchi, L.: An abstract model of service discovery and binding. *Formal Asp. Comput.* (to appear)
12. Fiadeiro, J.L., Lopes, A., Bocchi, L., Abreu, J.: The Sensoria reference modelling language. In: Wirsing and Hoelzl
13. Fu, X., Bultan, T., Su, J.: Conversation protocols: a formalism for specification and verification of reactive electronic services. *Theor. Comput. Sci.* 328(1-2), 19–37 (2004)
14. Goldblatt, R.: Logics of time and computation. CSLI, Stanford (1987)
15. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
16. Misra, J., Chandy, K.M.: Proofs of networks of processes. *IEEE Trans. Software Eng.* 7(4), 417–426 (1981)
17. OSOA. Service component architecture: Building systems using a service oriented architecture (2005), White paper, <http://www.osoa.org>
18. Reisig, W.: Towards a theory of services. In: Kaschek, R., Kop, C., Steinberger, C., Fliedl, G. (eds.) UNISCON 2008. LNBI, vol. 5, pp. 271–281. Springer, Heidelberg (2008)
19. Solanki, M., Cau, A., Zedan, H.: Introducing compositionality in web service descriptions. In: FTDCS, pp. 14–20. IEEE Computer Society, Los Alamitos (2004)
20. Su, J., Bultan, T., Fu, X., Zhao, X.: Towards a theory of web service choreographies. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 1–16. Springer, Heidelberg (2008)
21. Wirsing, M., Hoelzl, M. (eds.): Rigorous Software Engineering for Service-Oriented Systems. LNCS, vol. 6582. Springer, Heidelberg (2011)

rt-Inconsistency: A New Property for Real-Time Requirements

Amalinda Post¹, Jochen Hoenicke², and Andreas Podelski²

¹ Robert Bosch GmbH, Stuttgart, Germany

Amalinda.Post@de.bosch.com

² University of Freiburg, Germany

{hoenicke,podelski}@informatik.uni-freiburg.de

Abstract. We introduce rt-inconsistency, a property of real-time requirements. The property reflects that the requirements specify apparently inconsistent timing constraints. We present an algorithm to check rt-inconsistency automatically. The algorithm works via a stepwise reduction to real-time model checking. We implement the algorithm using an existing module for the reduction and the UPPAAL tool for the real-time model checking. As a case study, we apply our prototype implementation to existing real-time requirements for automotive projects at BOSCH. The case study demonstrates the relevance of rt-inconsistency for detecting errors in industrial real-time requirements specifications.

1 Introduction

The specification of requirements allows us to differentiate a *correct* from an *incorrect* system. Often, however, it is difficult to get the requirements specification itself right. In the case of real-time requirements, this difficulty is exacerbated by the presence of subtle dependencies between timing constraints.

A basic problem with getting the requirements right is the lack of unambiguous properties that allow us to differentiate a good from a bad set of requirements. The IEEE Standard 830-1998 of “Recommended Practice for Software Requirements Specifications” defines eight properties, called *correctness*, *unambiguity*, *completeness*, *consistency*, *ranking for importance*, *verifiability*, *modifiability*, and *traceability* [9]. The meaning of these properties is, however, not formally defined. To identify unambiguous properties for requirements remains an active research topic; see, e.g., [3,5,10,12].

In this paper, we propose a formal property of real-time requirements. The property reflects that the requirements specify apparently inconsistent timing constraints. Its violation may thus identify an (otherwise not identifiable) error in a requirements specification. We call the new property *rt-inconsistency* (for lack of a better name).

Errors in a requirement specification are often identified as *inconsistency* (the specification is unsatisfiable by any system, e.g., because it contains two contradicting requirements) or *incompleteness* (the specification lacks a requirement,

e.g., because one among several possible cases is not covered) [3]. The new property lies between inconsistency and incompleteness because the error can be repaired either by removing a requirement (as in the case of inconsistency) or by adding requirements (as in the case of incompleteness).

We demonstrate the relevance of the new property of real-time requirements specifications by a practical case study in an industrial setting. We took six existing sets of real-time requirements for automotive projects at BOSCH. Each of the six sets had undergone a thorough review. Yet, three out of the six sets of requirements contained an error identifiable through rt-inconsistency. The errors were acknowledged and subsequently repaired by the responsible engineers at BOSCH. In one of the three cases, this required a major revision of the requirements. The errors could not have been caught using the property of inconsistency; i.e., each of the six sets was consistent, as we could verify formally. We do not know of any existing property of requirement specifications that would have allowed us to catch these errors.

In the standard industrial praxis, requirements specifications must be checked manually, e.g., by peer reviews [12]. Yet, since requirements affect each other and cannot be analyzed in isolation, this is an considerable effort. Automatic checks are desirable already for small sets of requirements [6,7,13].

In this paper, we show that we can check the rt-inconsistency of a set of real-time requirements automatically. We present an algorithm and its theoretical foundation. The algorithm works via a stepwise reduction of the rt-inconsistency of a set of real-time requirements to a certain property of one specific real-time system (that we derive from the set of real-time requirements). I.e., it reduces a property of properties of real-time systems to a property of a real-time system.

The reduction allows us to reduce rt-inconsistency checking to *real-time model checking*. As a theoretical consequence of the reduction, the algorithm inherits the theoretical exponential worst-case complexity. Practically, the reduction allows us to capitalize on the advances of real-time model checking and the industrial strength of existing tools such as UPPAAL [2].

To implement the algorithm, we build upon pre-existing modules from [8] for deriving the real-time system from the set of given real-time requirements and the UPPAAL tool [2] for checking the real-time system. The implementation has allowed us to perform the above-mentioned case study with existing industrial examples. The primary goal was to demonstrate the practical relevance of the new property of real-time requirements. The second goal of the case study was to evaluate the practical potential of our algorithm for checking the property automatically. The results of our experiments are encouraging in this direction. They indicate that checking rt-inconsistency automatically is feasible in principle.

Roadmap. We will next illustrate rt-inconsistency informally with an example. Section 2 introduces rt-inconsistency formally, together with the formalization of real-time requirements. Section 3 presents the algorithm, with (1) the notion of automaton used for the intermediate step of the reduction, (2) the construction of such an automaton from requirements, and (3) its transformation to a timed automaton. Section 4 presents the case study.

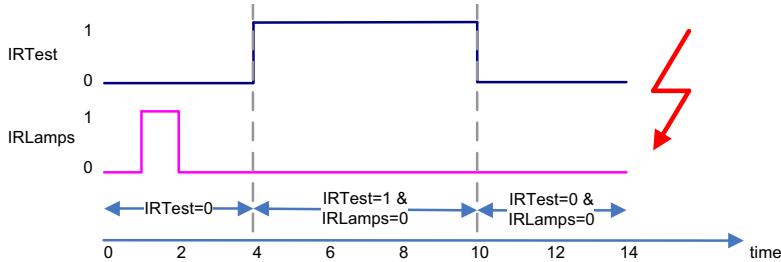


Fig. 1. Witness for the rt-inconsistency of the set of requirements $\{Req_1, Req_2\}$. The timing conflict appears immediately after the time point $t = 14$.

Example of rt-inconsistency. Consider the two informal real-time requirements below.

- Req_1 : “If the system’s diagnostic request $IRTest$ is set, then it is never the case that the infrared lamps stay turned off for more than 10 seconds.”
- Req_2 : “If the system’s diagnostic request $IRTest$ is set, then it is never the case that the infrared lamps will be on in the next 6 seconds.”

The set of the two requirements is consistent (one can find systems that satisfy both requirements). However, a closer inspection of the requirements shows that circumstances may arise where the two requirements are in conflict. Consider the trace depicted in Figure 1. At time point $t = 4$ the diagnostic request $IRTest$ is set. By Req_1 , the infrared lamps must be turned on within the next 10 seconds. In the (right-open) time interval $[4, 10)$ $IRTest$ stays. It disappears at the time point $t = 10$. By Req_2 , the infrared lamps are turned off for at least 6 seconds (and, thus, during the whole time interval $[10, 16]$). I.e., for any possible continuation of the trace after $t = 14$, the two requirements clash: by Req_1 , the infrared lamps are turned on, and by Req_2 , they are turned off for further two seconds; i.e., the requirements claim contradicting valuations for the infrared lamps. The set of the two requirements, while consistent, is rt-inconsistent!

One way to resolve the rt-inconsistency is to delete Req_2 or to change it to the *weaker* requirement Req'_2 .

- Req'_2 : “If the system’s diagnostic request $IRTest$ is set *and it was not set in the last 10 seconds*, then it is never the case that the infrared lamps will be on in the next 6 seconds.”

Another way to resolve the rt-inconsistency is to add both, the requirements Req_3 and Req_4 to the set containing Req_1 and Req_2 .

- Req_3 : “Once the system’s diagnostic request $IRTest$ is set, $IRTest$ stays active for at most 3 seconds.”
- Req_4 : “Once the system’s diagnostic request $IRTest$ disappears, $IRTest$ is absent for at least 10 seconds.”

2 Defining rt-Inconsistency

To find rt-inconsistencies we need to interpret requirements on both the infinite time axis $\mathcal{R}_{\geq 0}$ and on finite time intervals $[0, t]$ from zero to some time point t . A convenient way to obtain a suitable formalization of requirements is to borrow the notation of the *Duration Calculus* [14,15]. Before we introduce the formal syntax of our class of real-time requirements, we will derive the formalization of the example requirement *Req₁* from Section 1. We first restate *Req₁* in a less ambiguous form.

- *Req₁*: If the system’s diagnostic request *IRTest* is set *at a time when the infrared lamps are turned off*, then it is never the case that the infrared lamps stay turned off for more than 10 seconds.

We introduce the predicates *IRTest* and *IRLampsOn* (with their obvious meaning) and reformulate *Req₁* as follows.

- *Req₁*: For any run of the system, it must not be the case that there are time points t_1 , t_2 , and t_3 , $t_1 < t_2 < t_3$ such that *IRTest* is true between t_1 and t_2 , and *IRLampsOn* is false between t_1 and t_3 , and the length of the interval $[t_2, t_3]$ is greater than 10 seconds.

Equivalently, for any run of the system, it must not be possible to split the time axis into four consecutive *phases* where:

1. the first phase (from time point 0 to t_1) does not underlie any constraint,
2. the second phase (from time point t_1 to t_2) underlies the constraint that *IRTest* is true and *IRLampsOn* is false,
3. the third phase (from time point t_2 to t_3) underlies the constraint that *IRLampsOn* is false and the constraint that its length (the difference between t_3 and t_2) is greater than 10.
4. the fourth phase (from time point t_3 until infinity) does not underlie any constraint.

In formal syntax, the requirement *Req₁* is expressed as the formula φ_1 below. Here the symbol “ \neg ” denotes negation, the symbol “ $;$ ” separates two phases, the phase “[P]” refers to a nonzero-length period of time during which the predicate P is satisfied, adding the conjunct “ $\ell > k$ ” to a phase means that its length is strictly greater than the constant k , and the constant phase “*true*” refers to a period of time during which the behavior does not underlie any constraint (and which is possibly of zero length).

$$\varphi_1 = \neg(\text{true} ; [\text{IRTest} \wedge \neg\text{IRLampsOn}] ; [\neg\text{IRLampsOn}] \wedge \ell > 10 ; \text{true})$$

The formalization of three other requirements from Section 1 is given below.

$$\varphi_2 = \neg(\text{true} ; [\text{IRTest}] ; \text{true} \wedge \ell < 6 ; [\text{IRLampsOn}] ; \text{true})$$

$$\varphi_3 = \neg(\text{true} ; [\text{IRTest}] \wedge \ell > 3 ; \text{true})$$

$$\varphi_4 = \neg(\text{true} ; [\text{IRTest}] ; [\neg\text{IRTest}] \wedge \ell < 10 ; [\text{IRTest}] ; \text{true})$$

Syntax. Formally, the syntax of phases π and requirements φ is defined by the BNF below. The predicate symbol P refers to a fixed set Preds of predicate symbols (for *observations* whose truth values change over time). The correctness of the algorithm presented in this paper (more precisely, the soundness of the answer “rt-consistent”) relies on the fact that we have only *strict* inequalities ($\ell > k$ and $\ell < k$) in the definition of phases π . The extension to non-strict inequalities ($\ell \geq k$ and $\ell \leq k$) would complicate the algorithm unnecessarily, i.e., without being motivated by practical examples.

$$\begin{array}{ll} \text{phase} & \pi ::= \text{true} \mid [P] \mid \pi \wedge \ell > k \mid \pi \wedge \ell < k \\ \text{requirement} & \varphi ::= \neg(\pi_1 ; \dots ; \pi_n ; \text{true}) \end{array}$$

A set of requirements denotes their conjunction. We overload the metavariable φ for requirements and sets of requirements.

Interpretation \mathcal{I} . Avoiding the confusion about the different meanings of other terms in the literature, we use the term *interpretation* to refer to a mapping that assigns to each time point t on the time axis (i.e., each $t \in \mathcal{R}_{\geq 0}$) an observation, i.e., a valuation of the family of given predicates P .

$$\mathcal{I} : \mathcal{R}_{\geq 0} \rightarrow \{\text{true}, \text{false}\}^{\text{Preds}}, \quad \mathcal{I}(t)(P) \in \{\text{true}, \text{false}\}$$

We use “*segment* of \mathcal{I} from b to e ” and write “ $(\mathcal{I}, [b, e])$ ” for the restriction of the function \mathcal{I} to the interval $[b, e]$ between the (“begin”) time point b and the (“end”) time point e .

We use “*prefix* of \mathcal{I} until t ” for the special case of the segment of \mathcal{I} from 0 to t , i.e., for the restricted function $(\mathcal{I}, [0, t])$. Given two interpretations \mathcal{I} and \mathcal{I}' we say that the prefix of \mathcal{I} until t *coincides* with the prefix of \mathcal{I}' until t if the (restricted) functions are equal, i.e., $(\mathcal{I}, [0, t]) = (\mathcal{I}', [0, t])$.

Satisfaction of a requirement by an interpretation, $\mathcal{I} \models \varphi$. We first define the satisfaction of a requirement by a segment of an interpretation, $(\mathcal{I}, [b, e]) \models \varphi$.

$$\begin{aligned} (\mathcal{I}, [b, e]) \models [P] &\quad \text{if } \mathcal{I}(P)(t) \text{ is true for } t \in [b, e] \text{ and } b \neq e \\ (\mathcal{I}, [b, e]) \models \ell > k &\quad \text{if } (e - b) > k \\ (\mathcal{I}, [b, e]) \models \pi_1 ; \pi_2 &\quad \text{if } (\mathcal{I}, [b, m]) \models \pi_1 \text{ and } (\mathcal{I}, [m, e]) \models \pi_2 \text{ for some } m \in [b, e] \end{aligned}$$

We can then define the satisfaction of a requirement by a (‘full’) interpretation.

$$\mathcal{I} \models \varphi \quad \text{if } (\mathcal{I}, [0, t]) \models \varphi \text{ for all } t$$

That is, an interpretation \mathcal{I} satisfies the requirement φ if every prefix of \mathcal{I} does (i.e., if for every time point t , the prefix of \mathcal{I} until t satisfies φ).

Safety. A requirement φ , which we have defined to be a negated formula of the form $\varphi = \neg(\pi_1 ; \dots ; \pi_n ; \text{true})$, expresses that “something bad may not happen”, where “bad” refers to the possibility of splitting the time axis

into $n + 1$ intervals satisfying the phases π_1, \dots, π_n , and *true*, respectively. The requirement φ expresses a so-called *safety property* (which means that “if an execution violates φ , then there is a prefix of the execution such that any execution with this prefix violates φ ”). The syntactic restriction that the last phase is *true* is crucial here.

rt-inconsistency. The satisfaction of a requirement is defined not only for a ‘full’ interpretation on the infinite time axis (“ $\mathcal{I} \models \varphi$ ”) but also for the prefix of an interpretation until a time point t (“ $(\mathcal{I}, [0, t]) \models \varphi$ ”). We need both satisfaction relations in our definition of rt-inconsistency.

Definition 1. [rt-inconsistency] A set of requirements φ is *rt-inconsistent* if there exists a prefix $(\mathcal{I}, [0, t])$ of an interpretation \mathcal{I} until a time point t that satisfies φ but no extension of the prefix to a full interpretation does (i.e., on the whole time axis), formally:

$$\begin{aligned} (\mathcal{I}, [0, t]) &\models \varphi \\ \mathcal{I}' &\not\models \varphi \quad \text{if } (\mathcal{I}', [0, t]) = (\mathcal{I}, [0, t]) \end{aligned}$$

i.e., the full interpretation \mathcal{I}' does not satisfy φ whenever its prefix until t coincides with the prefix of the interpretation \mathcal{I} until t .

Remark 1. In essence, a set of requirements φ is rt-inconsistent if it does not exclude the existence of a prefix of an interpretation which leads to a conflict. The conflict prevents the possibility of the extension of the prefix to a full interpretation. More precisely, in the setting of Definition 1 (of an interpretation \mathcal{I} , a time point t , and the prefix $(\mathcal{I}, [0, t])$ satisfying φ), the rt-inconsistency is due to one of two reasons.

1. No extension of the prefix $(\mathcal{I}, [0, t])$ to any time point t' after t is possible without violating φ (i.e., $(\mathcal{I}, [0, t']) \not\models \varphi$ for all $t' > t$).

In other words: The conflict inherent in φ hits directly after t . The conflict does not hit at any time point in the closed interval $[0, t]$ but it does hit when time leaves the interval, i.e., at any time point t' after t . No passing of time after t is possible without a conflict.

2. There are (uncountably many) time points t' after t such that the extension of the prefix $(\mathcal{I}, [0, t])$ to t' is possible without violating φ , but there exists a time point t_0 after all those time points t' such that the extension of the prefix $(\mathcal{I}, [0, t])$ to t_0 violates φ (i.e., $(\mathcal{I}, [0, t_0]) \not\models \varphi$ for some $t_0 > t$).

In other words: The conflict inherent in φ strikes after the time point t_0 after t . The conflict does not hit at t or at any other time point in the right-open interval $[0, t_0)$ but it does hit at the time point t_0 . No passing of time into t_0 is possible without a conflict.

Remark 2. It is easy to find examples that show, respectively, that neither does the rt-consistency imply the absence of *deadlocks* for every system that satisfies the real-time requirement φ , nor does the rt-inconsistency imply the presence of deadlocks in every system that satisfies φ .

Remark 3. Although the idea of using rt-inconsistency to detect flaws in real-time requirements is new (and in particular no algorithm for deciding rt-inconsistency was given before), the property has already appeared in a different form in [1]. There, however, the concern is the completeness of proof methods for the treatment of real time in standard linear temporal logic. The goal in [1] is a method to prove that a system specification S satisfies a requirement φ . Here, S is an “old-fashioned” program where updates of a program variable *now* over the reals are used to model the progress of time. If φ is a liveness property, then one may need to add to S a fairness assumption NZ for the scheduler that updates *now*. Still, a proof method may be incomplete (i.e., it may be incapable of showing the correctness of S together with NZ wrt. the requirement φ). The completeness of a proof method may hold only for correctness problems where the pair (S, NZ) is *machine-closed*. Machine-closure of S for NZ is essentially the same as rt-consistency of S . The formal definitions are not directly comparable (since machine-closure is formalized using sequences of pairs of states and time points, as opposed to continuous interpretations \mathcal{I}).

3 Checking rt-Inconsistency

In this section, we present an algorithm (see Algorithm 1) to check whether a set of requirements φ is rt-inconsistent. To simplify the presentation, we assume that φ is consistent (we have implemented the check for consistency, not presented here, and use it in a preliminary step in our experiments).

As already explained in the introduction, we reduce the problem of checking the rt-inconsistency of φ to the problem of checking a certain temporal property (existence of *deadlocks*) of a real-time system, formally a *timed automaton* S . We construct S from φ . More precisely, we first construct a certain kind of automaton A , a so-called *phase event automaton* (PEA), from φ and then transform A into a timed automaton S such that φ , A , and S are related in a sense that we will make formal.

Fig. 2 presents the intermediate results of the different steps of the application of Algorithm 1 to the rt-inconsistent set of the requirements φ_1 and φ_2 from Section 2 (formalizing Req_1 and Req_2 from Section 1). Algorithm 1 transforms the requirements φ_1 and φ_2 into the phase event automata A_1 and A_2 in Figure 2a resp. Figure 2b. It forms their parallel product $A = A_1 \parallel A_2$ which is given in Figure 2c. It then transforms A into the timed automaton S given in Figure 2d. After that it checks whether S contains a deadlock. In this example, it finds a deadlock and returns the answer “ φ is rt-inconsistent”, together with a witness given in Figure 3 (a run of S leading to a deadlock). The first witness depicted in Figure 3 (where *IRTest* toggles too quickly) suggests adding the requirement φ_4 . The second witness (where *IRTest* stays on too long) suggests adding the requirement φ_3 .

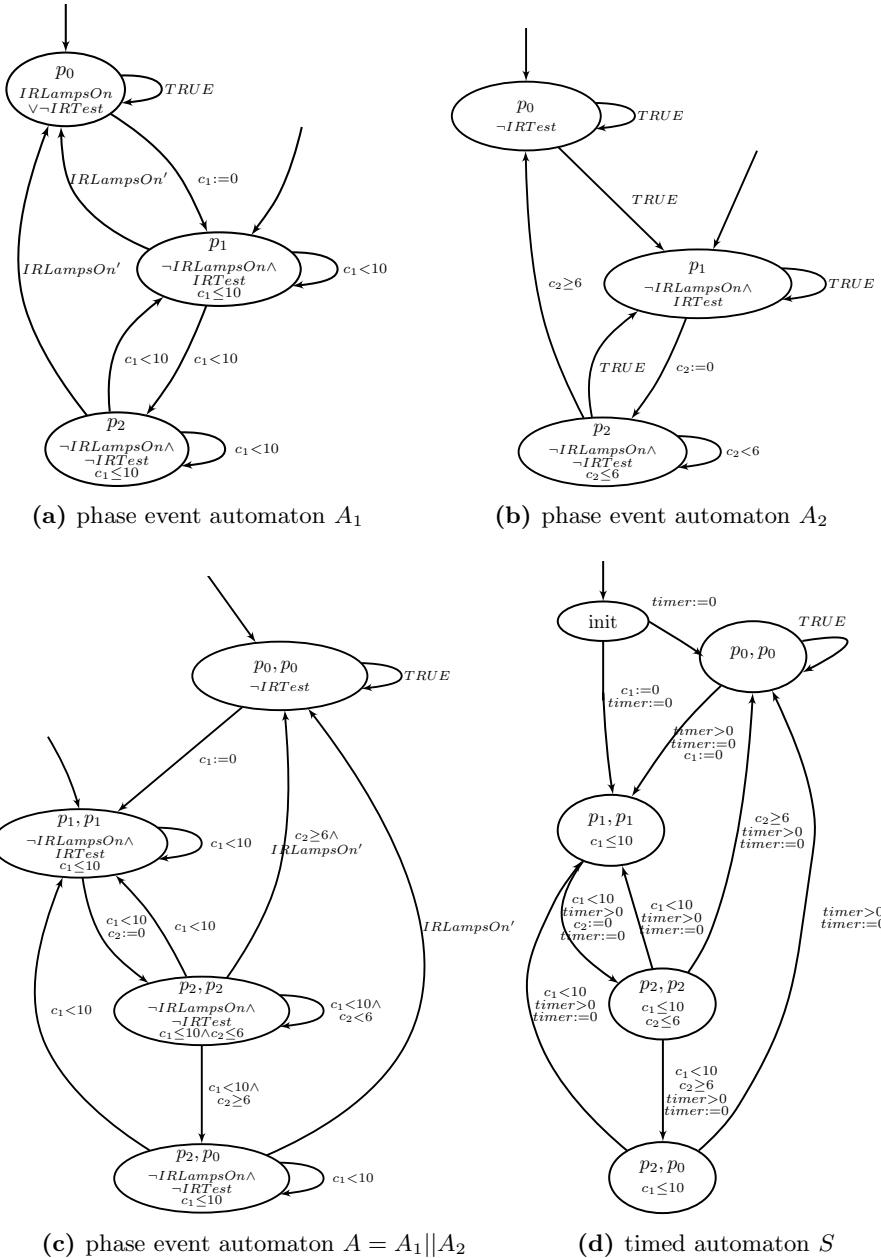


Fig. 2. Algorithm 1 of Section 3 applied to the set of the requirements φ_1 and φ_2 from Section 2 (formalizing Req_1 and Req_2 from Section 1) constructs the phase event automata A_1 and A_2 , forms their parallel product $A = A_1 \parallel A_2$ and transforms A into the timed automaton S .

$$\varphi_1 = \neg(\text{true} ; [\text{IRTest} \wedge \neg\text{IRLampsOn}] ; [\neg\text{IRLampsOn}] \wedge \ell > 10 ; \text{true})$$

$$\varphi_2 = \neg(\text{true} ; [\text{IRTest}] ; \text{true} \wedge \ell < 6 ; [\text{IRLampsOn}] ; \text{true})$$

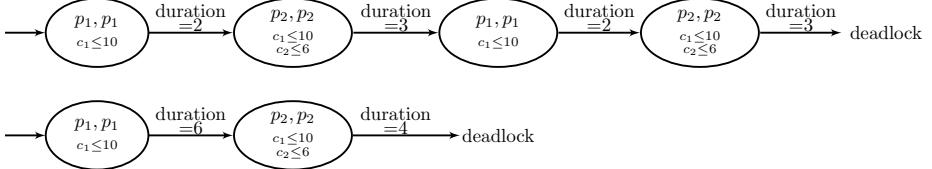


Fig. 3. Algorithm 1 of Section 3 applied to the set of the requirements φ_1 and φ_2 from Section 2 returns a run of S leading to a deadlock as a witness for the answer “ φ is rt-inconsistent”, e.g. one of the depicted runs. The depicted witnesses suggest adding φ_3 respectively φ_4 .

Algorithm 1. Check rt-inconsistency of set of requirements $\varphi = \{\varphi_1, \dots, \varphi_n\}$

```

for all  $i = 1, \dots, n$  do
     $A_i := \text{req2pea}(\varphi_i)$            {transform requirement to phase event automaton}
end for
 $A := A_1 \| \dots \| A_n$           {form the parallel product of phase event automata}
 $S := \text{pea2ta}(A)$              {transform phase event automaton to timed automaton}
                                {call timed model checker for existence of deadlocks}
if ( $S$  is deadlock-free) then
    return “ $\varphi$  is rt-consistent”
else
    return “ $\varphi$  is rt-inconsistent”   {return path to deadlock in timed automaton}
end if

```

3.1 Phase Event Automata

We will use *phase event automata* as a means to define sets of interpretations \mathcal{I} (i.e., mappings from time points to observations, i.e., to valuations of predicates). Syntactically, a phase event automaton resembles a timed automaton in that it has the same notion of *clocks*; semantically, there are differences such as in the minimal duration between transitions. Below, for a set of variables X , we use X' for the set of their primed versions (which stand, as usual, for the value of the corresponding variable in a successor state after a transition). We use $\mathcal{L}(X)$ to denote a set of formulae with free variables in X .

A *phase event automaton* (PEA) is a tuple $A = (P, V, C, E, s, I, P^0)$ where

- P is the set of locations p (*phases*),
- C is the set of clocks c ,
- V is the Boolean variables P (*observation predicates*),
- E is a set of *transitions* of the form (p, g, X, p') where p and p' specify the from- and to-locations, the guard g is a formula in the unprimed clock variables and in the unprimed and primed Boolean variables (i.e., g specifies also the updates of Boolean variables), and X is the set of clocks that are reset to 0, i.e., $E \subseteq P \times \mathcal{L}(C \cup V \cup V') \times 2^C \times P$,

- the mapping s assigns each location p its *state invariant* which is stated as a formula in the Boolean variables, i.e., $s : P \rightarrow \mathcal{L}(V)$,
- the mapping I assigns each location p its *clock invariant* which is stated as a formula in the clocks, more precisely a conjunction of inequalities $c \leq k$ or $c < k$ with $c \in C$ and $k \in \mathcal{R}_{\geq 0}$, i.e., $I : P \rightarrow \mathcal{L}(C)$,
- P^0 is the set of initial locations, i.e., $P^0 \subseteq P$.

We use *runs* to describe the operational semantics of a PEA. A run r is a (finite or infinite) sequence of quadruples (p, β, γ, t) consisting of a location p , a valuation of the Boolean variables $\beta : V \rightarrow \{\text{true}, \text{false}\}$, a valuation of the clocks $\gamma : C \rightarrow \mathcal{R}_{\geq 0}$, and a *non-zero duration* t (the amount of time spent in the location p), i.e., $t > 0$.

Given the PEA A of the form above, r is a run of A if it starts in an initial location with clock values 0, and for each quadruple (p, β, γ, t) in r , the valuation of variables β satisfies the state invariant of location p (i.e., $\beta \models s(p)$), the clock valuation γ satisfies the clock invariant at location p during the whole duration t (i.e., $\gamma + t \models I(p)$), and for each pair of consecutive quadruples (p, β, γ) and (p', β', γ') , the valuations satisfy the guard and the update constraint of a transition in E of the form (p, g, X, p') , i.e., $(\beta, \beta', \gamma + t) \models g$ (where β' is applied to the primed variables in g) and $\gamma'(c) = 0$ if c in X and $\gamma + t$ otherwise.

The *duration* of a run r is the sum of the durations t in its quadruples. An infinite run r is *non-Zeno* if its duration is infinite. An *unextendable run* of A is a finite run r of A which is not prefix of any non-Zeno run of A .

Interpretations accepted by A , $\mathcal{L}(A)$. A run r *matches* an interpretation \mathcal{I} if for *almost all* time points t , the value of \mathcal{I} coincides with the valuation β in the quadruple of r that corresponds to t if one adds up the durations of all quadruples in r preceding it. We omit the cumbersome formal definition (which is analogous for finite runs and prefixes of an interpretation).

An interpretation \mathcal{I} is *accepted* by A , formally $\mathcal{I} \in \mathcal{L}(A)$, if there is a non-Zeno run r of A that matches \mathcal{I} . The next lemma implies that every run r of A gives rise to an interpretation \mathcal{I} accepted by A .

Lemma 1. *For every non-Zeno run r of a phase event automaton A there exists an interpretation \mathcal{I} such that r matches \mathcal{I} .*

The prefix of the interpretation \mathcal{I} until the time point t is *accepted* by A , formally $(\mathcal{I}, [0, t]) \in \mathcal{L}(A)$, if there is a run r of A with duration t that matches $(\mathcal{I}, [0, t])$.

A phase event automaton A *represents* a requirement φ if it accepts exactly the interpretations that satisfy φ , i.e., $\mathcal{I} \in \mathcal{L}(A)$ if and only $\mathcal{I} \models \varphi$. Given two PEAs A_1 and A_2 representing the requirements φ_1 resp. φ_2 , their parallel product $A_1 \parallel A_2$ (defined in the canonical way) represents their conjunction $\varphi_1 \wedge \varphi_2$.

3.2 Characterizing rt-Inconsistency via Phase Event Automata

We will use the algorithm of [8,11] which, given a requirement φ , constructs a phase event automaton A that represents φ . In this section, we show that the

properties of the algorithm that are stated in Lemmas 2 and 3 (and proven in [8]) suffice to characterize the rt-inconsistency of φ . From now on, we refer to the construction of A from φ by the algorithm of [11].

Lemma 2. *The phase automaton A constructed from φ is deterministic; i.e., if A accepts the prefix of the interpretation until the time point t , then there is exactly one run r of A that matches \mathcal{I} for duration t .*

Lemma 3. *The prefix of the interpretation \mathcal{I} until the time point t satisfies the requirement φ if and only if it is accepted by the phase automaton A constructed from φ ; i.e., $(\mathcal{I}, [0, t]) \models \varphi$ if and only if $(\mathcal{I}, [0, t]) \in \mathcal{L}(A)$.*

The “ \Leftarrow ” direction of Lemma 3 relies on the restriction to *strict* inequalities in the definition of the syntax of φ in Section 2. The restriction entails that the PEA constructed from φ contains only non-strict clock invariants “ $c \leq k$ ”.

Theorem 1. *The set of requirements φ is rt-inconsistent if and only if the phase event automaton A constructed from φ contains an unextendable run.*

Proof. “ \Rightarrow ” If φ is rt-inconsistent and \mathcal{I} is an interpretation as in Definition 1, then the prefix of \mathcal{I} until the time point t satisfies φ and thus, by Lemma 3, it is accepted by A . Hence, by the definition of acceptance, there is a run r in A that matches \mathcal{I} for duration t . We are done if we show that r is unextendable. Assume, for a proof by contraction, that there is a non-Zeno run r' of A that extends r . By Lemma 1, r' matches an interpretation \mathcal{I}' . Hence, again by the definition of acceptance, A accepts \mathcal{I}' and also the prefix of \mathcal{I}' until t' , for every time point t' . By Lemma 3, the prefix of \mathcal{I}' until t' satisfies φ , for every time point t' . Thus, by the definition of the satisfaction relation, the full interpretation \mathcal{I}' satisfies φ . Since the prefix of \mathcal{I} until the time point t coincides with the one of \mathcal{I}' , we have found an interpretation \mathcal{I}' as in Definition 1, i.e., one which cannot exist.

“ \Leftarrow ” If r is an unextendable run of A for, say, the duration t , then, by Lemma 3, there is an interpretation \mathcal{I} such that the prefix of \mathcal{I} until t matches r . By Lemma 3, the prefix of the interpretation \mathcal{I} until the time point t satisfies the requirement φ . We are done if we show that there exists no interpretation \mathcal{I}' that satisfies φ and whose prefix until t coincides with the one of \mathcal{I} . Assume, for a proof by contraction, that such an \mathcal{I}' exists. Then, since A represents φ , A accepts φ . By the definition of acceptance, there exists a non-Zeno run r' of A that matches \mathcal{I}' . By Lemma 2, A is deterministic, i.e., r' coincides with r for the duration t , or: r is a prefix of r' . Thus, we have found a non-Zeno run of A which has r as a prefix, which cannot exist by the assumption that r is an unextendable run of A . \square

3.3 Characterizing rt-Inconsistency via Timed Automata

From phase event automata to timed automata. Algorithm 2 transforms a phase event automaton to a timed automaton. The transformation extends a similar one in [8] which preserves reachability but not unextendability. The transformation introduces a special clock *timer* in order to capture the fact that, in a

Algorithm 2. Transform phase event automaton A to timed automaton S

```

if  $A$  has more than one initial location  $p_{0i}$  then
    add a new initial location, with an transition to every  $p_{0i}$ 
end if
normalize transitions such that each guard is a conjunct of literals
for all transitions  $(p, g, X, p')$  of  $A$  do
    if  $s(p) \wedge g \wedge (s(p'))'$  is unsatisfiable then
        remove this transition
    end if
end for
remove unreachable locations
remove all literals from the guards except clock constraints
set all state invariants to true
for all transitions  $(p, g, X, p')$  of  $A$  do
     $g := g \wedge \text{timer} > 0$ 
     $X := X \cup \{\text{timer}\}$ 
    for all constraints  $c \leq k$  in  $I(p')$  where  $c \notin X$  do
         $g := g \wedge c < k$ 
    end for
end for

```

phase event automaton, every location getting active has to stay active for a non-zero period of time. The clock *timer* is reset when the location is entered, and every outgoing transition must satisfy the guard that specifies $\text{timer} > 0$. To prevent introducing artificial deadlocks, the new transformation strengthens the guard of the outgoing transition with the strict inequality $c < k$ derived from the non-strict inequality $c \leq k$ in the clock invariant of the target location (thus, after a transition, there is always some time left to stay in the target location).

Lemma 4. *Every run of the phase event automaton A corresponds to a run of the timed automaton S constructed from A (with the same sequence of locations and clock valuations), and vice versa.*

Deadlock. Following [2], a timed automaton S contains a *deadlock* if there is a reachable state (p, γ) such that for all durations $d > 0$ there is no action successor of $(p, \gamma + d)$. In particular, the self-loop is not enabled. We will next characterize rt-inconsistency in terms of deadlocks, and thus obtain the correctness of Algorithm 1.

Theorem 2 (Correctness). *The timed automaton S constructed from the set of requirements φ via the phase event automaton A and Algorithm 2 contains a deadlock if and only if the set of requirements φ is rt-inconsistent.*

Proof. “ \Rightarrow ” By Lemma 4, a run to a deadlock state (p, γ) in S corresponds to an unextendable run in A to the same location p with the same clock valuation γ without any further transition being possible (the successor would be reachable in S as well). By Theorem 1, φ is rt-inconsistent.

“ \Leftarrow ” We assume that S is deadlock-free and show that φ is not rt-inconsistent. By Theorem 1 it suffices to show that, for every finite run r of A , there is a non-Zeno run r' of A that extends r .

By Lemma 4, a finite run r of A corresponds to a run in the timed automata S leading to some state (p, γ) . We extend r by staying in p until the bound of its invariant is reached (the clock invariants in S are non-strict by the construction). If p has no bound, r' can be chosen as the non-Zeno run which stays in p forever. Otherwise, since S contains no deadlock, there must be an action successor from p , at a different location whose time bound is not yet reached. The infinite iteration of this reasoning leads to an infinite run in the timed automaton and, again by Lemma 4, to an infinite run r' in the PEA A . We need to show that r' is non-Zeno.

Assume that there exists a time point, say, t , such that r' never reaches t . If b is the smallest among the bounds of all clocks in the invariant of some location in S , then each location can be visited at most t/b times in r' (since the location is active until the clock reaches the bound and then the clock must be reset before the location can be visited again). Since there are only finitely many locations in S , r' is a finite run. This is in contradiction to our construction of r' . Hence there is no such time point t and r' is a non-Zeno run. \square

4 Using rt-Inconsistency in a Case Study

The goal of our experimental study is to evaluate the practical relevance of rt-inconsistency. The primary question we need to investigate is whether the property is useful to improve a requirement specification, namely by providing a criterion that helps to differentiate good from bad, or desirable from undesirable requirement specifications. According to our preliminary results, this is indeed the case; see Table 1.

Table 1 refers to six examples from different automotive projects at BOSCH. Each example is a set of real-time requirements for a single software component. The specifics of the components are not relevant; hence we do not present them and just number the examples from 1 to 6 (first column). The second column refers to the number of requirements in the example. Each requirement specification had previously undergone a thorough albeit informal review. We formalized the requirements (i.e., we translated them to formal requirements as defined in Section 2) in a somewhat lengthy process of iterations with feedback from the responsible requirement engineers. We had the final formalization reviewed by a requirements engineer.

As Table 1 shows, three out of the six examples have an error that is identifiable as rt-inconsistency. I.e., for Components 1, 2, and 3, the rt-inconsistency identifies an actual flaw in the requirement specification that needed to be repaired. As the last column shows, major changes were needed to correct the requirement specification. E.g., for Component 3, two of the existing requirements were deleted, five were changed, and seven new requirements were added. If positive, the rt-inconsistency test returns a run of the time automaton that

Table 1. Checking rt-inconsistency for existing examples of sets of real-time requirements for software components in automotive projects at BOSCH using a prototypical implementation (Fig. 3) of the algorithm presented in Section 2, on a PC Windows XP system with 2GHz Intel Core 2 Duo processor and 1GB RAM, whereas only one core was used. The examples are numbered from 1 to 6. The columns refer to: the size of the input in the number of requirements, the number of nodes resp. the (much higher!) number of transitions of the timed automaton (TA) obtained by the automatic translation of the set of requirements (via the translation to a phase event automaton), the time used for the automatic translation, the time used by the timed model checker UPPAAL for checking the existence of deadlocks in the timed automaton (where n/a here means out-of-memory when loading the input), the outcome of the rt-inconsistency check, and the cost of the *correction* of an rt-inconsistency (in the number of requirements that were newly added (A), changed (C), and deleted (D), respectively).

	reqs	TA nodes	TA transitions	$\{\text{reqs}\} \mapsto \text{TA}$	UPPAAL	rt-consistent?	correction
1	9	900	69183	34s	37s	no	A:3, C:4
2	10	2520	418365	322s	28min 49s	no	A:4, C:4
3	10	895	36541	9.4s	1h 16min	no	A:7, D:2, C:5
4	13	28	310	1s	< 1s	yes	—
5	16	27	729	6s	< 1s	yes	—
6	17	1614	318267	160s	n/a	n/a	n/a

is helpful for analyzing the error; yet, debugging the requirement specification demands the help of a requirements engineer with domain knowledge and takes a considerable amount of time (about half a day to one day per example for debugging and fixing). Most of the detected flaws based on conflicts similar to the conflict described in the example of Section 1.

The requirement specifications for Component 4 and Component 5 are examples for rt-consistency. Without these examples one might wonder if rt-inconsistency implies a high degree of specificity (obtainable only through a large number of precise requirements) which cannot be found in realistic examples. The examples, requirement specifications that are rt-consistent *as is* (i.e., not after a revision), indicate that this is not the case. For safety-critical systems (e.g., in the automotive domain), the high degree of specificity enforced by rt-consistency seems appropriate.

In order to evaluate the practical relevance of rt-inconsistency, the second question we need to investigate is whether the property can be checked on realistic examples automatically. For the purpose of proof of concept, we have implemented the algorithm presented in Section 3; see Figure 4. Our prototypical, non-optimized implementation relies on existing tool kits [8,2] for implementing three procedures called by the algorithm: the translation of a set of requirements to a phase event automata (req2pea), the translation of a phase event automaton into a timed automaton (pea2ta), the check for the existence of deadlocks in a timed automaton. The results of our experiments (see Table 1) show that in five out of six examples, the algorithm is able to automatically prove resp. disprove rt-inconsistency. The examples are relatively small but they are realistic (and

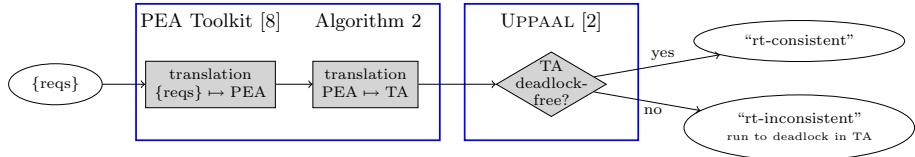


Fig. 4. Prototype implementation of Algorithm 1 for checking rt-inconsistency of a set of requirements, with modules using tools for phase event automata (PEA) resp. timed automata (TA)

apparently so complex that a manual review is no longer sufficient). The results indicate that checking rt-inconsistency automatically is feasible in principle.

As one could expect by the theoretical complexity of the algorithm, the check does not succeed for every input (and, as often with the automatic analysis tools, the size of the input does not necessarily correlate with the difficulty of its analysis for the tool). In the sixth example, UPPAAL runs out of memory when loading the timed automaton generated from the phase event automaton in this example. We still need to analyze the cause (which is not solely the size of the input), but it is clear that UPPAAL is not optimized for the timed automata generated in this setting; in the examples of Table 1, the number of transitions is two orders of magnitude larger than the number of nodes.

An experimental study is incomplete (and somewhat unsatisfying) if it does not expose deficiencies of the evaluated concepts (and opportunities for improvement). The sixth example shows that the state explosion problem occurs not only in theory, but also in practice.

Furthermore, the case study shows that the state explosion is not directly related to the number of requirements: although Component 4 and Component 5 consist of more requirements than the first three components the number of nodes and transitions of A is much smaller for Component 4 and 5. This is due to the fact that not all requirements blow up the state space exponentially. There are also requirements that solely constrain the state space and thus reduce the number of states, e.g., a requirement like “If $IRTTest$ holds, then $Diagnosis.Running$ holds as well” forbids every state in which holds $IRTTest \wedge \neg Diagnosis.Running$.

5 Conclusion and Future Work

We have introduced rt-inconsistency, a new property of requirements for real-time systems. We have shown that it has an interesting practical potential for unambiguously identifying subtle timing errors in a requirements specification. We have presented an algorithm to check rt-inconsistency automatically. We have implemented the algorithm to demonstrate its feasibility *in principle*, by applying it to prove the absence resp. presence of rt-inconsistency in a number of existing requirement specifications in automotive projects. Our experiments

discovered previously unknown errors in some of those specifications, errors which got subsequently repaired.

As already mentioned, one line of future work is to adapt heuristics and optimizations from real time model checking to checking rt-inconsistency. Another, more speculative line of research are methods to automatically correct (or help to correct) an rt-inconsistent set of requirements, possibly using algorithms from *real-time synthesis* [4].

References

1. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, pp. 1–27. Springer, Heidelberg (1992)
2. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL (2004)
3. Dahlstedt, A.G., Persson, A.: Requirements interdependencies - moulding the state of research into a research agenda. In: REFSQ, pp. 71–80 (2003)
4. Ehlers, R., Mattmüller, R., Peter, H.-J.: Combining symbolic representations for solving timed games. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 107–121. Springer, Heidelberg (2010)
5. Hayes, J.H.: Building a requirement fault taxonomy: Experiences from a NASA verification and validation research project. In: ISSRE (2003)
6. Heimdahl, M.P.E., Leveson, N.G.: Completeness and consistency analysis of state-based requirements. IEEE Trans. on SW Engineering, 3–14 (1995)
7. Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G.: Automated consistency checking of requirements specifications. ACM Trans. on SW Eng. and Methodology 5(3), 231–261 (1996)
8. Hoenicke, J.: Combination of Processes, Data, and Time. PhD thesis, University of Oldenburg (July 2006)
9. IEEE. Recommended Practice for Software Requirements Specifications (1998)
10. Leveson, N.G.: System safety in computer-controlled automotive systems. In: SAE World Conference (2000)
11. Meyer, R., Faber, J., Hoenicke, J., Rybalchenko, A.: Model checking duration calculus: a practical approach. Formal Asp. Comput. 20(4-5), 481–505 (2008)
12. Walia, G.S., Carver, J.C.: A systematic literature review to identify and classify software requirement errors. Inf. Softw. Technol. 51(7), 1087–1109 (2009)
13. Yu, L., Su, S., Luo, S., Su, Y.: Completeness and consistency analysis on requirements of distributed event-driven systems. In: TASE (2008)
14. Zhou, C., Hansen, M.R.: Duration Calculus: A Formal Approach to Real-Time Systems. Springer, Heidelberg (2004)
15. Zhou, C., Hoare, C., Ravn, A.: A calculus of durations. In: IPL (1991)

Automatic Flow Analysis for Event-B^{*}

Jens Bendisposto and Michael Leuschel

Institut für Informatik, Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
`{bendisposto,leuschel}@cs.uni-duesseldorf.de`

Abstract. In Event-B a system is developed using refinement. The language is based on a relatively small core; in particular there is only a very small number of substitutions. This results in much simpler proof obligations, that can be handled by automatic tools. However, the downside is that, in case of software development, structural information is not explicitly available but hidden in the chain of refinements. This paper discusses a method to uncover these implicit algorithmic structures and use them in a model checker. Other applications are code generation, model comprehension, and test-case generation.

Keywords: Event-B, Model Checking, Theorem Proving, Tool Integration.

1 Introduction

Some specification formalisms only have limited ways to express ordering of events. In particular Event-B [1] lacks a notion of sequential composition, or other ways to explicitly describe the ordering of events. If we specify software or systems that include software in Event-B, we often have some implicit algorithmic structure.¹ Unfortunately this information is implicit only and therefore not directly usable by tools nor directly visible to users. This paper discusses a method to uncover this implicit algorithmic structure. This information can be useful for analyzing or comprehending models and for automatic code generation. In this paper we also show how to use this information to improve model checking.

The paper is structured as follows: After introducing some Event-B notions in section 2, we discuss in section 3 the dependency relation between events. Section 4 shows how to compute the so-called enabling predicates, and sections 5 and 6 demonstrate the exploitation of these predicates for model checking. In sections 7 and 8 we show how we can construct a flow from the enabling predicates, revealing the implicit algorithmic structure of the model. Finally, we discuss applications and restrictions of the method and some related work.

^{*} This research is being carried out as part of the DFG funded research project GEPAVAS.

¹ To order events in Event-B the usual method is to introduce abstract program counters.

2 Preliminaries

We follow the style of [1] of expressing variables and substitution in formulas. In particular, let $v = v_1, \dots, v_n$ be a sequence of n distinct variables, $t = t_1, \dots, t_n$ a sequence of n formulas and F a formula. Then $F[t/v]$ is obtained from F by replacing simultaneously all free occurrences of each v_i by t_i . We let $F(v)$ denote a formula, whose free variables are among v_1, \dots, v_n . Once the formula $F(v)$ has been introduced, we denote by $F(t)$ the formula $F[t/v]$ with v replaced by t .

In Event-B a state consists of a set of variables that are modified by events. The values of the variables are constrained by invariants $I(v)$. Each event is composed of a *guard* $G(t, v)$ and an *action* $S(t, v)$, where t are *parameters* of the event. We will only consider events of the form

$$\begin{aligned} \text{evt} &\stackrel{\Delta}{=} \text{any } t \\ &\quad \text{when } G(t, v) \\ &\quad \text{then } v_{i_1}, \dots, v_{i_k} := E_1(v, t), \dots, E_k(v, t) \text{ end} \end{aligned}$$

for some $i_j \in i_1, \dots, i_n$. Note that t can be empty and $G(t, v)$ can be *true*. Also note that k can be 0, in which case we write the action part as *skip*.

All assignments of an action $S(t, v)$ occur simultaneously. Variables v_{j_1}, \dots, v_{j_l} that do not appear on the left-hand side of an assignment of an action are not changed by the action. The effect of an assignment can be described by a before-after predicate:

$$S(v, t, v') \stackrel{\Delta}{=} v'_{i_1} = E_1(v, t) \wedge \dots \wedge v'_{i_k} = E_k(v, t) \wedge v'_{j_1} = v_{j_1} \wedge \dots \wedge v'_{j_l} = v_{j_l}$$

A before-after predicate describes the relationship between the state just before an assignment has occurred, x , and the state just after the assignment has occurred, x' .

Note that Event-B also allows non-deterministic actions of the form $x : \in E(t, v)$ or $x : | Q(t, v, x')$. Without loss of generality, we assume that those are rewritten to the above form using new parameters, one for every non-deterministic action which denotes the chosen element. For instance, we rewrite

$$\text{any } max \text{ when } max > 10 \text{ then } x : \in 1..max \text{ end}$$

into

$$\text{any } max, choice \text{ when } max > 10 \wedge choice : 1..max \text{ then } x := choice \text{ end}$$

3 Dependency between Events

We are interested in how events influence each other. The motivations are multiple: either we may try to understand the dynamic behavior of our model, we may wish to generate code by determining the control flow or we may wish to improve the performance of model checking.

Suppose we have an event g with action $x, y := (x + 1), 0$. There are various ways it can influence another event:

1. it can disable another event. E.g., the event h with guard $y > 0$ will for sure be disabled after executing g .
2. it can enable another event. E.g., the event h' with guard $y = 0$ would for sure be enabled after executing g .
3. it can be independent of another event. For example, the enabling of the event h'' with guard $z > 0$ would not be modified by executing g , i.e., it will be enabled after g if and only if it was enabled before. (Note that, depending on the action part of h'' , the effect of h'' could have been modified.)

In cases 1 and 2 the enabling or disabling may depend on the current state of the model. Take for example the event h''' with guard $y = 0 \wedge x > 1$. Then h''' would be enabled after g if $x > 0$ holds in the state before executing g , and disabled otherwise. The predicate $x > 0$ is what we call an enabling predicate, and which we define as follows:

Definition 1 (Enabling predicate). *The predicate P is called enabling predicate for an event h after an event g , denoted by $g \rightsquigarrow_{P(v,t,s)} h$, if and only if the following holds*

$$I(v) \wedge G(v, t) \wedge S(v, t, v') \Rightarrow (P(v, t, s) \Leftrightarrow H(v', s))$$

where $I(v)$ is the invariant of the machine, $G(v, t)$ is the guard of g with parameters t and $S(v, t, v')$ the before-after predicate of its action part, and where $H(v, s)$ is the guard of h with parameters s .

In the absence of non-deterministic actions, an equivalent definition can be obtained using the weakest precondition notation:

$$I(v) \wedge G(v, t) \Rightarrow (P(v, t, s) \Leftrightarrow [S(t, v)]H(v, s))$$

where $[S]P$ denotes the weakest precondition which ensures that after executing the action S the predicate P holds.

Note that it is important for us that the action part $S(t, v)$ of an event does not contain any non-determinism (i.e., that all non-determinism has been lifted to the parameters t ; see Section 2). Indeed, in the absence of non-determinism, the negation of an enabling predicate is a disabling predicate, i.e., it guarantees that the event h is disabled after g if it holds (together with the invariant) before executing g . However, if we have non-determinism the situation is different. There may even exist no solution for $P(v, t, s)$ in Def. 1, as the following example shows.

Example 1. Take $x : \{1, 2\}$ as the action part of an event g with no parameters and the guard $true$ and $x = 1$ as the guard of h . Then $[S(t, v)]H(v) \equiv false$ as there is no way to guarantee that h is enabled after g . Indeed, there is no predicate over x that is equivalent to $x' = 1$ in the context Def. 1 : the before after predicate $S(v, t, v')$ is $x' \in \{1, 2\}$ and does not link x and x' . Similarly, there is no way to guarantee that h is disabled after g . In particular, $\neg[S(t, v)]H(v) \equiv true$ is not a disabling predicate.

Note that if $I(v) \wedge G(v) \wedge [S(t, v)]H(v, s)$ is inconsistent, then any predicate $P(v, t, s)$ is an enabling predicate, i.e., in particular $P(v, t, s) \equiv \text{false}$.

How can we compute enabling predicates? Obviously, $[S(t, v)]H(v)$ always satisfies the definition of an enabling predicate. What we can do, is simplify it in the context of $I(v) \wedge G(v)$.² We will explain later in Sect. 4 how we compute enabling predicates and discuss the requirements for a simplifier.

Example 2. Take for instance a model of a for loop that iterates over an array and increments each value by one. Assuming the array is modeled as a function $f : 0..n \rightarrow \mathbb{N}$ and we have a global counter $i : 0..(n + 1)$, we can model the for loop (at a certain refinement level) using two events *terminate* and *loop*.

$$\text{terminate} \hat{=} \text{when } i > n \text{ then skip end}$$

$$\text{loop} \hat{=} \text{when } i \leq n \text{ then } f(i) := f(i) + 1 || i := i + 1 \text{ end}$$

We can now try to find enabling predicates for each possible combination of events. Table 1 shows the proof obligations from Def. 1 and simplified predicates P which satisfy it.

Table 1. Enable Predicates for a simple model

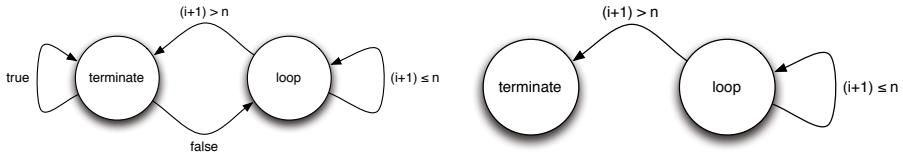
Event Pairs (first \rightsquigarrow_P second)	Enable Predicate Definition (wp notation)	Simplified P
terminate \rightsquigarrow_P terminate	$i > n \implies (P \iff i > n)$	true
loop \rightsquigarrow_P loop	$i \leq n \implies (P \iff (i + 1) \leq n)$	$(i + 1) \leq n$
loop \rightsquigarrow_P terminate	$i \leq n \implies (P \iff (i + 1) > n)$	$(i + 1) > n$
terminate \rightsquigarrow_P loop	$i > n \implies (P \iff i \leq n)$	false

The directed graph on the left in Figure 1 is a graphical representation of Table 1. Every event is represented by a node and for every enabling predicate $\text{first} \rightsquigarrow_P \text{second}$ from Table 1 there is an edge between the corresponding nodes.

The right picture shows the same graph if we take independence of events into account, i.e., if an event g cannot change the guard of another event h , we do not insert an edge between g and h . In particular, as *terminate* does not modify any variables, it cannot modify the truth value of any guard. On first sight it seems as if we may have also lost some information, namely that after the execution of *terminate* the event *loop* is certainly disabled. We will return to this issue later and show that for the purpose of reducing model checking and other application, this is actually not relevant.

In Event-B models of software components independence between events occurs very often, e.g., if an abstract program counter is used to activate a specific subset of the events at a certain point in the computation. We can formally define independence as follows.

² This is similar to equivalence preserving rewriting steps within sequent calculus proofs, where $I(v), G(v)$ are the hypotheses and $[S(t, v)]H(v)$ is the goal of the sequent.

**Fig. 1.** Graph Representations of Dependence for a Simple Model

Definition 2 (Independence of events). Let g and h be events. We say that h is independent from g — denoted by $g \not\rightsquigarrow h$ — if the guard of h is invariant under the substitution of g , i.e., iff the following holds:

$$I(v) \wedge G(v, t) \wedge S(v, t, v') \implies (H(v, s) \iff H(v', s))$$

Our first observation is that an event g can only influence the enabledness of an event h (we do not require $g \neq h$) if g modifies some variables that are read in the guard of h . We denote the set of variables used in the guard of h by $\text{read}(h)$ and the set of variables modified by g by $\text{write}(g)$. If $\text{write}(g)$ and $\text{read}(h)$ are disjoint, then h is trivially independent from g :

Lemma 1. For any two events h and g we have that $\text{read}(h) \cap \text{write}(g) = \emptyset \Rightarrow g \not\rightsquigarrow h$.

This happens in our loop example, because $\text{write}(\text{terminate}) = \emptyset$, and hence all events (including terminate itself) are independent from terminate .

However, $\text{read}(h) \cap \text{write}(g) = \emptyset$ is sufficient for independence of events but not necessary. Take for instance the events from Figure 2. Event g clearly modifies variables that are read by h and therefore $\text{read}(h) \cap \text{write}(g) \neq \emptyset$ but g can not enable or disable h .

<pre> event g begin x := x + 1 y := y - 1 end </pre>	<pre> event h when x + y > 5 then end </pre>
--	---

Fig. 2. Independent events

The trivial independence can be decided by simple static analysis, i.e., by checking if $\text{read}(h) \cap \text{write}(g) = \emptyset$. Non trivial independence is in general undecidable. In practice, it is a good idea to try to prove that two events are independent in the sense of Def. 2, as it will result in a graph representation with fewer edges. However, it is not crucial for our method that we detect all independent events.

As we have seen in the right side of Fig. 1, the information we gain about enabling and independence can be represented as a directed graph, now formally defined as follows.

Definition 3 (Enable Graph). An Enable Graph for an Event-B model is a directed edge labeled graph $G = (V, E, L)$. The vertices V of the graph are the events of the model. Two events can be linked by an edge if they are not independent, i.e., $(g \mapsto h) \notin E \Rightarrow g \not\sim h$. Each existing edge $g \mapsto h$ is labeled with the enabling predicate, i.e., $g \sim_{L(g \mapsto h)} h$.

Above we define a family of enable graphs, depending on how precise our information about independence is. Below, we often talk about the enable graph for a model, where we assume a fixed procedure for computing independence information.

Aside. There is another representation of the graph that is sometimes more convenient for human readers. We can represent the graph as a forest where each tree has one event as its root and only the successor nodes as leafs. The alternate representation is shown in Figure 3 for a small example.

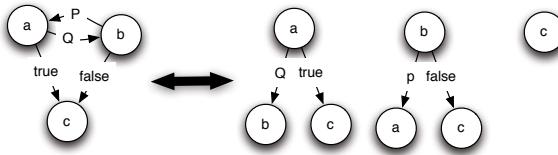


Fig. 3. Representations of the Enable Graph

4 Computing the Enabling Predicates

As mentioned before the weakest precondition $[S(t, v)]H(v, t, s)$ obviously satisfies the property of enabling predicates. We can use the syntax tree library of the Rodin tool to calculate the weakest precondition. However, these candidates for enabling predicates need to be simplified, otherwise they are as complicated as the original guard and we will gain no benefit from them. Therefore we simplify the candidate, in the context of the invariant $I(v)$ and the guard of the preceding event $G(v, t)$.

Consider the model shown in Figure 4. The weakest precondition for g preceding h is $[x := x + 2]x = 1$ which yields $x = -1$. This contradicts the invariant $x > 0$ and thus h can never be executed after g took place. In the context of the invariant, $x = -1$ is equivalent to false.

The simplification of the predicates is an important step in our method, deriving an enabling predicate $P(v, t, s)$ from the weakest precondition $[S(t, v)]H(v, s)$. Recall, that we simplify the predicate $[S(t, v)]H(v, s)$ in the context of the invariant $I(v)$ (and the guard $G(v, t)$)

$$I(v) \wedge G(v, t) \Rightarrow (P(v, t, s) \Leftrightarrow [S(t, v)]H(v, s))$$

A very important requirement in our setting is that the simplifier never increases the number of conjuncts. We have to keep the input for our enable and flow graph

```

invariant x > 0
event g           event h
begin             when x = 1
  x := x + 2
end

```

Fig. 4. Simplification

constructions small to prevent exponential blowup. Our simplifier shall find out if a conjunct is equivalent to true or false. In the first case the conjunct can be removed from the predicate in the second case the whole predicate is equivalent to false.

We have implemented a prototype simplifier in Prolog that uses a relatively simple approach. This prototype was used to carry out our case studies. The method does not rely on this implementation, we can replace it by more powerful simplification tools in the future.

5 Using the Enable Graph for Model Checking

The enable graph contains valuable information for a model checker. In this section we describe how it can be used within PROB. When checking the consistency of an Event-B model, PROB traverses the state space of the model starting from the initialization and checks the model's invariant for each state it encounters. The cost for checking a state is the sum of the cost of evaluating the invariant for the state and the calculation of the successors. Finding successor states requires to find solutions for the guards of each event. A solution means that the event is applicable and we can find some parameter values. PROB then applies the actions to the current state using the parameter values resulting in some successor states. In some cases the enable graph can be used to predict the outcome of the guard evaluation. The special case of an enabling predicate $P = \text{false}$ is very important. It means that no matter how we invoke g we can omit the evaluation of the guard of h because it will be false after observing g . In other words it is a proof that the property h is disabled holds in any state that is reachable using g .

When encountering a new state s via event e , we look up e in the enable graph. We can safely skip evaluation of the guards of all events f that have an edge (e,f) which is labeled with *false* in the Enabled Graph. We can even go a step further if we have multiple ways to reach s . When considering an event to calculate successor states we can arbitrary choose one of the incoming events and use the information from the enable graph. For instance, if we have four events a, b, c and d and we know that a disables c and b disables d . Furthermore we encounter a state s via a but do not yet calculate the successors. Later we encounter s again, this time via b . When calculating the successors we can skip both, c and d .

The reason is that we have a proof for *c is disabled* because the state was reachable using event *a* and a proof that *d is disabled* because the state was reachable using event *b*. Thus the conjunction *c and d are disabled* is also true.

Because we use the invariant when simplifying the enabling predicate (see Section 4), the invariant must hold in the previous state in order to use the flow information. However we believe this is reasonable because most of the time we are hunting bugs and thus we stop at a state that violates the invariant. The implementation must take this into account and in case of an invariant violation it must not use the information gained by flow analysis. Also it needs to check not only the invariant but also the theorems if they are used in the simplifier.

6 Enable Graph Case Study

In this section, we will apply the concept to a model of the extended GCD algorithm taken from [7] using our prototype. The model consists of a refinement chain, where the last model consists of two loops. The first loop builds a stack of divisions. The second loop calculates the result from this stack. The last refinement level contains five events excluding the initialization. The events *up* and *dn* are the loop bodies, the events *upini*, *dnini* initialize the loops and *gcd* is the end of the computation. The event *init* is the *INITIALISATION* of the model.

Table 2. Read and write sets

event	read(event)	write(event)
init	\emptyset	$\{a, b, d, u, v, up, f, s, t, q, r, uk, vk, dn, dk\}$
upini	$\{up\}$	$\{up, f, s, t, q, r\}$
up	$\{up, r, f, dn\}$	$\{f, s, t, r, q\}$
gcd	$\{up, f, dn\}$	$\{d, u, v\}$
dnini	$\{up, dn, r, f\}$	$\{dn, dk, uk, vk\}$
dn	$\{dn, f\}$	$\{uk, vk, f\}$

The first step is to extract the read and write sets for each event; the result is shown in Table 2. Then we construct the enable graph. We calculate the weakest precondition for each pair of independent events and simplified them. Both steps were done manually but they were not very difficult. For instance, the most complicated weakest precondition was $[S_{up}]G_{dnini}$. In the presentation below we left out all parts of the guard and substitution that do not contain shared identifiers, e.g., the guard contains $up = \text{TRUE}$ but the substitution does not modify *up*. The next step is calculating the weakest precondition mechanically, finally we simplify the relational override using the rule $(r \Leftarrow a \mapsto b)(a) = b$.

$$\begin{aligned}
 [S_{up}]G_{dnini} &= [f := f + 1, r \Leftarrow \{f + 1 \mapsto f(t) \text{ mod } r(f)\}] (r(f) = 0) \\
 &= (r \Leftarrow \{f + 1 \mapsto t(f) \text{ mod } r(f)\})(f + 1) = 0 \\
 &= t(f) \text{ mod } r(f) = 0
 \end{aligned}$$

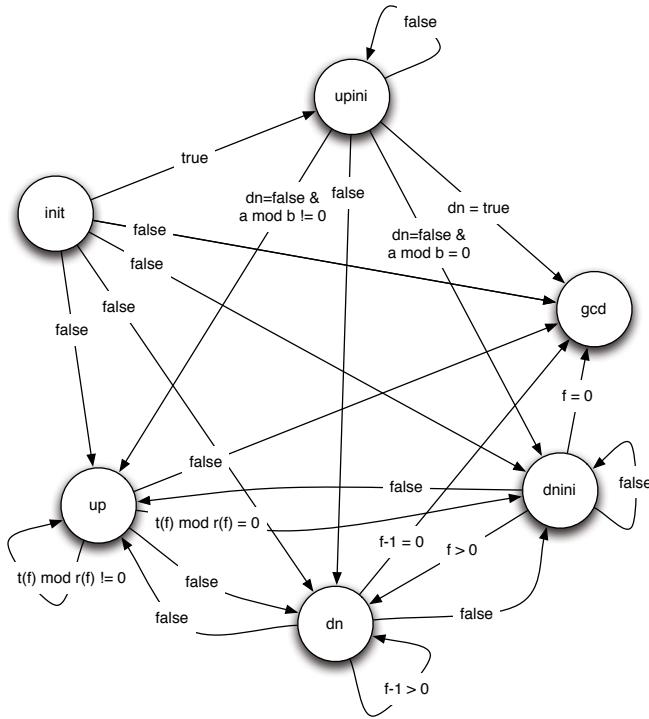


Fig. 5. Enable graph of the extended GCD example

The other simplification were much easier, for example, replacing $dn = \text{TRUE} \wedge dn = \text{FALSE}$ by *false*. The constructed graph is shown in Figure 5.

The enable graph can be used by the model checker to reduce the number of guard evaluations. Let us examine one particular run of the algorithm for fixed input numbers. The run will start with *init* and *upini* then contain a certain number of *up* events, say n . This will be followed by *dnini* and then exactly n *dn* events and will finish with one *gcd* event. In all, the calculation takes $2n + 4$ steps. After each step, the model checker needs to evaluate 5 event guards (one for each event, except for the guard of the initialization which does not need to be evaluated) yielding $10n + 20$ guard evaluations in total. Using the information of the enable graph we only need a total of $4n + 4$ guard evaluations. For example, after observing *up*, we only need to check the guards of *up* and *dnini*: they are the only outgoing edges of *up* in Fig. 5 which are not labelled by *false*.

7 Flow Construction

Beside the direct use in PROB, the enable graph can be used to construct a flow. A flow is an abstraction of the model's state space where an abstract state

represents a set of concrete states. Each abstract state is characterized by a set of events, representing all those concrete states where those (and only those) events are enabled.

A flow describes the implicit algorithmic structure of an Event-B model. This information is valuable for a number of different applications, such as code generation, test-case generation, model comprehension and also model checking. In Section 8 we illustrate how we can gain and exploit knowledge about a model using the flow graph. We also briefly discuss how to generate code based on the flow.

The flow graph is a graph where the vertices are labeled with sets of events, i.e., the set of enabled events. The edges are labeled with an event and a predicate composed from the enable predicates for this event. The construction of the flow graph takes the enable graph as its input. Starting from the state where only the initialization event is enabled the algorithm unfolds the enable graph. We will describe the unfolding in a simple example, an algorithm is shown in Figure 7 and 8.

Figure 6 shows a simple flow graph construction. On the left side the enable graph for the events *init*, *a* and *b* are shown. The graph reveals that *b* always disables itself while it does not change the enabledness of *a*. The event *a* keeps itself enabled if and only if *P* holds and it enables *b* if and only if *Q* holds. The *init* event enables *a* and disables *b*.

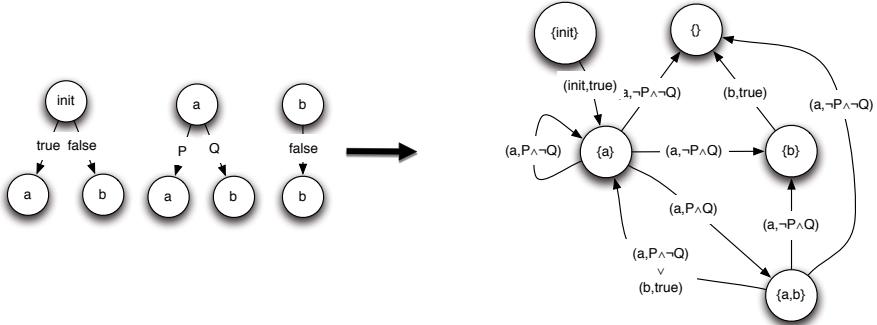


Fig. 6. Simple Flow Graph Construction

We start the unfolding in the state labeled with $\{\text{init}\}$. In this case we do not have a choice but to execute *init*. From the enable graph on the left hand side we know that after *init* occurs *a* is the only enabled event. Therefore we have to execute *a*. We know that if *P* is true then *a* will be enabled afterwards and analogously if *Q* holds then *b* will be enabled. Combining all combination of *P* and *Q* and their negations, we get the new states $\{\}$, $\{b\}$ and $\{a, b\}$. If we continue, we finally get the graph shown on the right hand side. If more than one event is enabled, we add edges for each event separately. We can combine edges by disjunction of the predicates. In our case we did that for the transition from $\{a, b\}$ to $\{a\}$ which can be used by either executing *b* or *a*.

The algorithm in Figure 8 calculates for a given event e the successors in the flow graph by combining all possible configurations. The algorithm also uses a list of independent events that are enabled in the current state and therefore they are also enabled in any new state. The algorithm in Figure 7 produces the flow graph starting from the state $\{init\}$.

Generating the Flow Graph can be infeasible because the graph can blow up exponentially in the numbers of events. However, in cases where constructing the flow graph is feasible, we gain a lot of information about the algorithmic structure and we can generate code if the model is deterministic enough. We will discuss applicability and restrictions of the methods in section 9.

```

 $todo := \{\{init\}\}$ 
 $done := \emptyset$ 
 $flow := \emptyset$ 
while  $todo \neq emptyset$  do
    choose  $node$  from  $todo$ 
    foreach  $e \in node$  do
         $keep := node \cap independent(e)$ 
         $atoms := expand(e, keep)$ 
         $todo := (todo \cup ran(atoms))$ 
         $flow := flow \cup \{node \mapsto atoms\}$ 
    od
     $done := done \cup \{node\}$ 
     $todo := todo - done$ 
od

```

Fig. 7. Algorithm for constructing a Flow Graph

Given: enable graph as $EG : (Events \times Events) \rightarrow Predicate$

```

def  $expand(e, keep) =$ 
     $true\_pred := \{f \mapsto true | (e \mapsto f) \in dom(EG) \wedge EG(e \mapsto f) = true\}$ 
     $maybe\_pred := \{f \mapsto p | (e \mapsto f) \in dom(EG) \wedge EG(e \mapsto f) = p \wedge p \neq false\}$ 
     $result := \emptyset$ 
    foreach  $s \subseteq node$  do
         $targets := dom(true\_pred) \cup dom(s) \cup keep$ 
         $predicate := \bigwedge ran(s) \wedge \neg(\bigvee ran(s) \triangleleft maybe\_pred)$ 
         $result := result \cup \{predicate \mapsto targets\}$ 
    od
    return  $result$ 
end def

```

Fig. 8. Algorithm for expanding the Enable Graph (i.e., computing successor configurations)

8 Flow Graph Case Study

If we apply the flow construction to the example graph shown in figure 5 we get the flow graph shown in Figure 10. Compared to the structured model developed by Hallerstede in [7] shown in 9 we see a very similar shape.

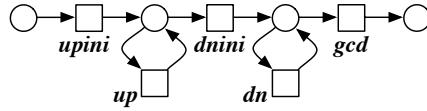


Fig. 9. Structural model from [7]

However, the automatic flow analysis helped us to discover an interesting property. The flow graph contains a state that corresponds to concrete states where no event is enabled, i.e., states where the system deadlocks. Thus the model contains a potential deadlock. Inspection showed that the deadlock actually does not occur. The reason why the flow graph contains the deadlock state is a guard that is too strong. The guards of dn and gcd only cover $f \geq 0$. The invariant implicitly prevents the system from deadlocking by restricting the values of f .

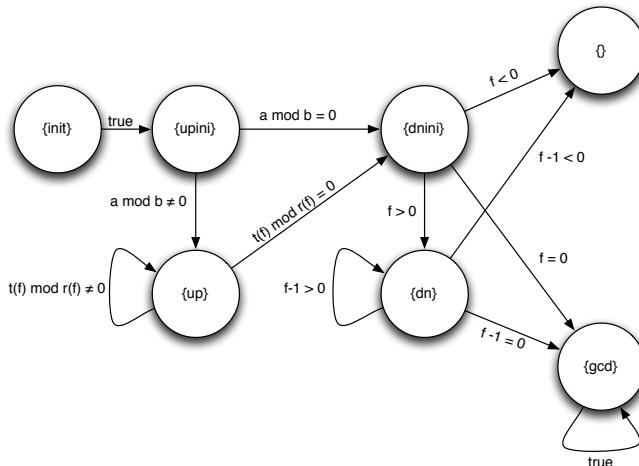


Fig. 10. Example for a relation between abstract and concrete states

In Figure 10 we can see that it is possible to automatically generate sequential code from a flow graph. The events up and dn can be translated into while loops and $upini$ and $dnini$ are *if – then – else* statements. In the particular case the termination of the computation was encoded into the gcd event.

9 Applicability and Restrictions

An important question is when to apply a method and maybe even more important when not to apply it. It is clear that flow analysis is probably not applicable if the model does not contain an algorithmic structure. In the worst case for flow construction, any combination of events can be enabled in some state, leading to $2^{card(Events)}$ states, where $card(Events)$ is the number of events. However in case of software developments it is very likely that eventually the model will contain events that are clustered, i.e., at each point during the computation a hopefully small set of events is enabled. We conjecture that the more concrete a model is, the better are results from simplification.

Constructing the enable graph is relatively efficient; it requires to calculate $O(card(Events)^2)$ enabling predicates. In case of software specifications generating the enable graph and using the information gained for guard reduction is probably worth trying. We can also influence the graph interactively. For instance, suppose the enable graph contains an edge labeled with $card(x) > 0$. Suppose we know that after the first event $x = \emptyset$ but the simplifier was too weak to figure it out, i.e., the empty set is written down in a difficult way, let's say $x := S \cap T$ where S and T are disjoint. By specifying (and proving) a theorem that helps the simplifier, e.g., $x = \emptyset$, we can interactively improve the graph. We believe that expressing these theorems does not only improve the graph but also our understanding of a model because we explicitly formalize properties of the model that are not obvious (at least not for the automatic simplifier).

Constructing the flow graph is much more fragile; it can blow up very fast. It is crucial to inspect the enable graph and try to reduce the size of the predicates as much as possible. However our experience is that Event-B models of software at a sufficient low level of refinement typically have some notion of an abstract program counter that implicitly control the flow in a model. These abstract program counters are not very complicated and therefore it is likely that they are exploited by the simplifier.

10 Related and Future Work

Inferring Flow Information. Model checking itself explores the state space of a model, and as such infers very fine-grained flow information. For Event-B, the PROB model checker [9,10] can be used for that purpose. However, it is quite rare that the complete state space of a model can be explored. When it is possible, the state space can be very large and flow information difficult to extract. Still, the work in [11] provides various algorithms to visualize the state space in a condensed form. The signature merge algorithm from [11] merges all states with the same signature, and as such will produce a picture very similar to the flow graph. However, the arcs are not labelled by predicates and the construction requires prior traversal of the state space.

Specifying Flow Information. There is quite a lot of related work, where flow information is provided explicitly by the modeler (rather than being deduced au-

tomatically, as in our paper). For example, several works use CSP to specify the sequencing of operations of B machines [14,3,5] or of Z specifications [6,12,13,2].

In the context of Event-B, there are mainly three other approaches that are related to our flow analysis. Hallerstede introduced in [7] a new approach to support refinement in Event-B that contains information about the structure of a component. Also Butler showed in [4] how structural information can be kept during refinement of a component. Both approaches have the advantage to incorporate the information about structure into the method, resulting in better precision. However both methods require the developer to use the methods from the beginning while automatic flow analysis can be applied to existing projects. In particular automatic flow analysis can actually be used to discover properties of a model such as liveness and feasibility of events. Hallerstede's structural refinement approach does not fully replace our automatic flow analysis. Both methods overlap to some extent, but we think that they can be combined, such that the automatic flow analysis uses structural information to ease the generation of the flow graph. In return, our method can suggest candidates for the intermediate predicates used during structural refinement.

The third approach is yet unpublished but implemented as a plug-in for Rodin [8]. It allows the developer to express flow properties for a model and to verify them using proofs.

Future Work. The next step is to fully integrate our method into the next release of PROB, and use it to improve the model checking procedure and help the user in analyzing or comprehending models. We also plan to use the technique to develop a new algorithm for test-case generation. In [15] we have introduced a first test-case generation algorithm for Event-B, tailored towards event coverage. One issue is that quite often it is very difficult to cover certain events. Here the flow analysis will hopefully help guide the model checker towards enabling those difficult events. We will also evaluate simplification tools that could be used within PROB to calculate good enabling predicates.

Conclusion. In summary, we have developed techniques to infer algorithmic structure from a formal specification. From an Event-B model, we have derived the enable graph, which contains information about independence and dependence of events. This graph can be used for model comprehension and to improve model checking. We have described a more sophisticated flow analysis, which derives a flow graph from an Event-B model. It can again be used for model comprehension, model checking but also for code generation.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Basin, D.A., Olderog, E.-R., Sevinç, P.E.: Specifying and analyzing security automata using csp-oz. In: Bao, F., Miller, S. (eds.) ASIACCS, pp. 70–81. ACM, New York (2007)

3. Butler, M.: csp2B: A practical approach to combining CSP and B. *Formal Aspects of Computing* 12, 182–198 (2000)
4. Butler, M.: Decomposition structures for event-B. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 20–38. Springer, Heidelberg (2009)
5. Butler, M., Leuschel, M.: Combining CSP and B for specification and property verification. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 221–236. Springer, Heidelberg (2005)
6. Fischer, C.: Combining object-z and CSP. In: Wolisz, A., Schieferdecker, I., Rennoch, A. (eds.) FBT, GMD-Studien, vol. 315, pp. 119–128. GMD-Forschungszentrum Informationstechnik GmbH (1997)
7. Hallerstede, S.: Structured Event-B Models and Proofs. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 273–286. Springer, Heidelberg (2010)
8. Iliašov, A.: Flows Plug-In for Rodin, http://wiki.event-b.org/index.php/Flows#Flows_plugin
9. Leuschel, M., Butler, M.: ProB: A model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
10. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. *STTT* 10(2), 185–203 (2008)
11. Leuschel, M., Turner, E.: Visualizing larger states spaces in ProB. In: Treharne, H., King, S., Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 6–23. Springer, Heidelberg (2005)
12. Mahony, B.P., Dong, J.S.: Blending object-z and timed csp: An introduction to tcoz. In: ICSE, pp. 95–104 (1998)
13. Smith, G., Derrick, J.: Specification, refinement and verification of concurrent systems—an integration of object-z and csp. *Formal Methods in System Design* 18(3), 249–284 (2001)
14. Treharne, H., Schneider, S.: How to drive a B machine. In: Bowen, J.P., Dunne, S., Galloway, A., King, S. (eds.) B 2000, ZUM 2000, and ZB 2000. LNCS, vol. 1878, pp. 188–208. Springer, Heidelberg (2000)
15. Wieczorek, S., Kozyura, V., Roth, A., Leuschel, M., Bendisposto, J., Plagge, D., Schieferdecker, I.: Applying Model Checking to Generate Model-Based Integration Tests from Choreography Models. In: Núñez, M., Baker, P., Merayo, M.G. (eds.) TESTCOM/FATES 2009. LNCS, vol. 5826, pp. 179–194. Springer, Heidelberg (2009)

Semantic Quality Attributes for Big-Step Modelling Languages

Shahram Esmaeilsabzali and Nancy A. Day

Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1

w t oo

Abstract. A semantic quality attribute of a modelling language is a desired semantic characteristic that is common to all models specified in that language. A modeller can enjoy the luxury of not having to model the invariants of the behaviour that are implicitly enforced by the semantic quality attributes. In this paper, we introduce three semantic quality attributes for the family of big-step modelling languages (BSMLs). In a BSML, a model's reaction to an environmental input is a sequence of small steps, each of which can consist of the execution of a set of transitions from multiple concurrent components. Each of our three semantic quality attributes specifies a desired property about how the sequence of small steps form a big step. We systematically enumerate the range of BSML semantics that satisfy each semantic quality attribute.

1 Introduction

Often when modelling a software system, there are many alternative languages that could be used. To narrow the range of alternatives, a modeller needs to answer the question of why language *A*, and not language *B*, is a more appropriate choice in a certain context. In this paper, we consider this question for the class of Big-Step Modelling Languages (BSMLs) [7, 8] from a semantic point of view.

BSMLs are a class of state-transition modelling languages that are suitable for modeling systems that interact with their environments continuously. In a BSML model, the reaction of a system to an environmental input is modelled as a big step that consists of a sequence of small steps, each of which can be the execution of a set of transitions from multiple concurrent components.¹ Examples of BSMLs are statecharts [10, 18], its variants [2], Software Cost Reduction (SCR) [11, 12], and the un-clocked variants of synchronous languages [9], such as Esterel [4] and Argos [17]. The variety of semantics for events, variables, and control states introduce a range of semantics for BSMLs. Previously, we *deconstructed* the semantics of this family of languages into a set of high-level, orthogonal variation points, which we call semantic aspects, and enumerated the common semantic options of each semantic aspect [7, 8].

A BSML provides a modeller with the convenience of describing the reaction of a system to an environmental input as the execution of a set of transitions, facilitating the

¹ The terms *macro step* and *micro step* are related to our big step/small step terminology. We use our own terminology to avoid connotation with a fixed semantics associated with these terms.

decomposition of a model into concurrent components. However, it also introduces the complexity of dealing with the semantic intricacies related to the *ordering* of these transitions, making it difficult at times to recognize the global properties of these models.

A *semantic quality attribute* of a modelling language is a desired semantic characteristic that is common to all models specified in that language. Thus, a modeller can enjoy the luxury of not having to model the invariants of the behaviour that are enforced by the semantic quality attribute. Our first contribution in this paper is to introduce three semantic quality attributes for BSMLs. Two BSMLs can be compared and distinguished based on their semantic quality attributes. Each semantic quality attribute exempts modellers from worrying about some of the complications of ordering in the sequence of the small steps of a big step. The *priority consistency* attribute guarantees that higher priority transitions are chosen over lower priority transitions. The *non-cancelling* attribute guarantees that if a transition becomes executable during a big step, it remains executable, unless it is executed. The *determinacy* attribute guarantees that all possible orders of small steps in a big step have the same result.

Each of our semantic quality attributes for BSMLs is a cross-cutting concern over our semantic aspects for BSMLs. Our second contribution in this paper is to specify the subsets of BSML semantics that satisfy each of the semantic quality attributes. For each semantic quality attribute, we identify necessary and sufficient constraints over the choices of the semantic options in our deconstruction that result in a BSML semantics that has the semantic quality attribute. In previous work [7, 8], we analyzed the advantages and disadvantages of each of the semantic options individually, to provide rationales to choose one over another. The analysis of the semantic quality attributes in this paper reveals interrelationships among seemingly independent semantic options. It also provides rationales for language design decisions that otherwise would have seemed ad hoc. For example, the specification of non-cancelling BSML semantics highlights the role of concurrency in small-step execution, while the specification of priority-consistent and determinate BSML semantics highlights the role of limiting the number of transitions that each concurrent component of a model can execute in a big step. A language designer or a modeller can either (i) use the semantic quality attributes to narrow the range of semantic options for a language, or (ii) gain insights about a language's attributes after choosing its semantic options.

Compared to related work, we introduce three novel semantic quality attributes for a broader range of modelling languages than considered previously. A notable example of introducing semantic quality attributes is Huizing and Gerth's work on the semantics of BSMLs that support only events. They introduced three semantic quality attributes for the semantics of events, therefore they only needed to deal with one semantic aspect and not cross-cutting concerns. Similar semantic properties to our semantic quality attributes have been considered in primitive formalisms [5, 14, 15, 16, 19], which are models of computations rather than practical languages. These efforts characterize a class of models that satisfy a property whereas we determine the set of BSML semantics that satisfy each semantic quality attribute. For example, our non-cancelling semantic quality attribute is similar to *persistence* for program schemata [15] and for Petri Nets [16].

The remainder of the paper is organized as follows. Section 2 presents an overview of the common syntax and semantics of BSMLs, together with an overview of BSML

semantic aspects and semantic options. Section 3 formally presents the semantic quality attributes together with the specification of the subsets of BSML semantics that satisfy each of the semantic quality attributes. Section 4 discusses related work. Section 5 concludes the paper.

2 Background: Big-Step Modelling Languages (BSMLs)

In this section, we present an overview of our deconstruction of the semantics of BSMLs. Section 2.1 presents the common syntax of BSMLs. Section 2.2 presents the common semantics of BSMLs, with Section 2.3 describing its semantic variation points through our deconstruction into semantic aspects and their semantic options. In our framework, an existing BSML is modelled by translating its syntax into the normal form syntax and its semantics into a set of semantic options. Our previous work contains a more comprehensive and a formal treatment of these concepts [6, 7, 8].

2.1 BSML Syntax

We use a normal form syntax to model the syntax of many BSMLs. In our normal form syntax, a BSML model is a graphical, hierarchical, extended finite state machine, consisting of: (i) a hierarchy tree of control states, and (ii) a set of transitions between these control states. In this section, we adopt a few syntactic definitions from Pnueli and Shalev’s work [18].

Control states. A **control state** is a named artifact that a modeller uses to represent a noteworthy moment in the execution of a model. A control state has a **type**, which is one of *And*, *Or*, or *Basic*. The set of control states of a model form a hierarchy tree. The leaves of the **hierarchy tree** of a model, and only they, are *Basic* control states. In a hierarchy tree, an *And* or an *Or* control state has a set of **child** control states. A control state is a **descendant** of another control state if it is its child through transitivity. Similarly, the **parent** and **ancestor** relations are defined. Two control states **overlap** if they are the same or one is an ancestor of the other. The children of an *And* control state are separated by dashed lines graphically. One of the children of an *Or* control state is its **default** control state, which is signified by an arrow without a source. Fig. 1 is an example BSML model that we use for illustration. The model specifies the behaviour of a system that controls the safety of an entrance to an industrial area. The system is either in the *Automatic* or in the *Manual* control state. The children of control state *Automatic* are *Or* control states *Temp*, *Lock*, and *Fan*. The default control state of *Temp* is *Low*, which is a *Basic* control state. Control state *Off* is one of the descendants of the *Automatic* control state. The **least common ancestor** of a set of control states is the lowest (closest to the leaves) control state in the hierarchy tree such that each of the control states is its descendant; e.g., the least common ancestor of *Locked* and *Unlocked* is *Lock*. Two control states are **orthogonal** if neither is an ancestor of the other and their least common ancestor is an *And* control state; e.g., *Locked* and *Off* are orthogonal.

Transitions. Each transition, t , has a **source control state**, $src(t)$, and a **destination control state**, $dest(t)$, together with the following four optional elements: (i) a **guard condition** (GC), $gc(t)$, which is a boolean expression over a set of variables, enclosed by a “[]”; (ii) a **triggering condition**, $trig(t)$, which is the conjunction of a set of events

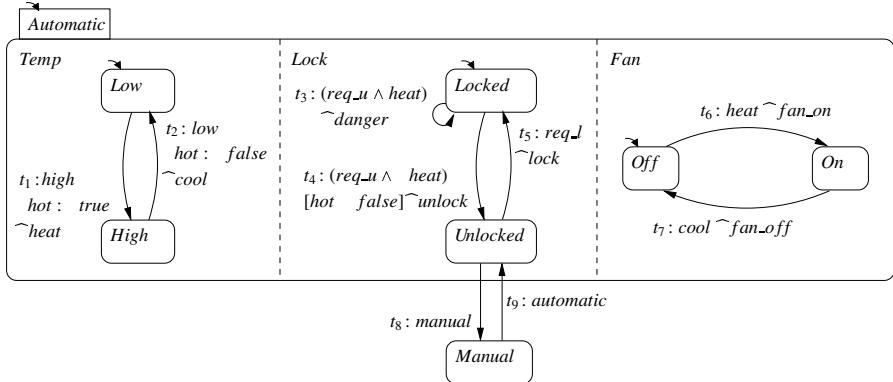


Fig. 1. Example: Industrial entrance control system

and negation of events; (iii) a set of **variable assignments**, $asn(t)$, which is prefixed by a “ $:$ ”, with at most one assignment to each variable; and (iv) a set of **generated events**, $gen(t)$, which is prefixed by a “ \sim ”. For example, in the model in Fig. 1, t_4 is a transition, with $src(t_4) = Locked$, $dest(t_4) = Unlocked$, $gc(t_4) = (hot \quad false)$, $asn(t_4) =$, $trig(t_4) = (req_u \wedge \neg heat)$, and $gen(t_4) = unlock$. When a set is a singleton, we drop its curly brackets; e.g., $asn(t_1) = hot : true$. The **scope** of a transition is the least common ancestor of its source and destination control states. The **arena** of a transition is the lowest *Or* control state that is an ancestor of its source and destination control states. The **root** of the hierarchy tree of a model, which is not always explicitly shown, is an *Or* control state so that the arena of each transition is defined. For example, in the model in Fig. 1, the scope of t_4 is *Lock* and the arena of t_8 is the root *Or* control state not shown in the figure. Two transitions are **orthogonal** if their source control states are orthogonal, as well as, their destination control states; e.g., t_4 and t_6 . A transition, t , is an **interrupt for** another transition, t' , if the sources of the transitions are orthogonal and one of the following conditions holds: (i) the destination of t' is orthogonal with the source of t , and the destination of t is not orthogonal with the sources of either transitions; or (ii) the destination of neither transition is orthogonal with the sources of the two transitions, but the destination of t is a descendant of the destination of t' . For example, in the model in Fig. 1, t_8 is an interrupt for t_6 , according to condition (i).

2.2 Common Semantics of BSMLs

A BSML model describes the continuous interaction of a system with its environment. A BSML model reacts to an environmental input through a **big step**, which consists of an alternating sequence of snapshots and **small steps**, with each small step consisting of the execution of a set of transitions. An **environmental input** consists of a set of events and assignments from the environment that are used throughout a big step. In the model in Fig. 1, events *high*, *low*, *req_u*, *req_l*, *manual*, and *automatic* are environmental input events. A **snapshot** is a collection of **snapshot elements**, each of which maintains information about an aspect of computing a big step. There is a snapshot element that maintains the set of current control states of a model: If a model resides in an *And*

control state, it resides in all of its children; if a model resides in an *Or* control state, it resides in one of its children, by default in its default control state. The execution of a small step causes a set of control states in the snapshot element to be **exited** and a set of control states to be **entered**. These sets are computed based on the scopes of the transitions in a small step. There are other snapshot elements for variables, events, etc.

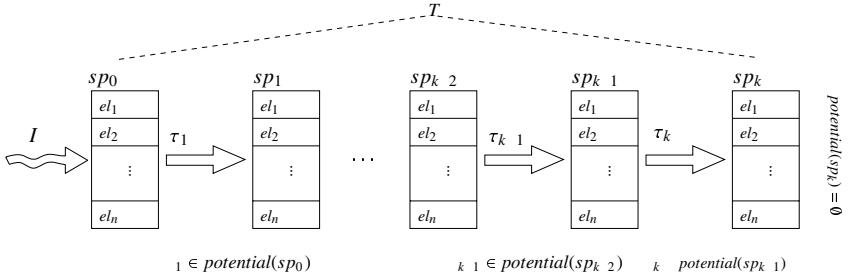


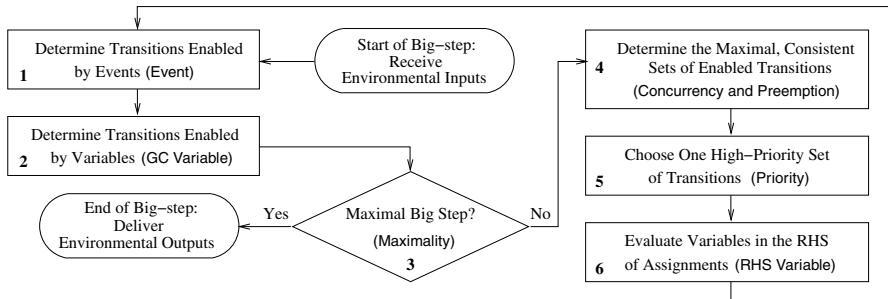
Fig. 2. Big step $T = \langle I \ sp_0 \ \tau_1 \ sp_1 \ \dots \ \tau_k \ sp_k \rangle$

Fig. 2 depicts the structure of a big step, which relates a source snapshot, sp_0 , and an environmental input, I , with a destination snapshot, sp_k . The effect of receiving I , which is captured in snapshot sp_0 , generates a sequence of small steps until there are no more small steps to be executed. Function *potential* specifies the set of all **potential small steps** at a snapshot, one of which is non-deterministically chosen as the next small step. Each big step or small step has a **source snapshot** and a **destination snapshot**. Formally, we represent a big step, T , as a tuple, $T = \langle I \ sp_0 \ \tau_1 \ sp_1 \ \dots \ \tau_k \ sp_k \rangle$. We use the accessor functions, *length*, *smallsteps*, and *trans*, to access the number of small steps of a big step, the set of sets of transitions representing the small steps of the big step, and the set of all transitions executed by the small steps of the big step, respectively. We use “.” to access an element of a tuple. As an example, for a big step, T , $T.sp_i$, where $1 \leq i \leq length(T)$, is the destination snapshot of its i th small step. For a BSML model M , we denote **all its possible big steps** as $bigsteps(M)$. This set includes all possible big steps in response to all environmental inputs at all possible snapshots. In our examples, we refer to a big step via its constituent sequence of sets of transitions.

2.3 BSML Semantic Variations

Fig. 3a, adapted from [7], describes our deconstruction of the semantics of BSMLs into six stages that each corresponds to a semantic variation point that affects how a big step is created. We call these variation points **semantic aspects**, and the possible variations of each its **semantic options**. Table 3b, which we will refer to throughout the rest of the paper, lists our semantic aspects and their semantic options, together with a brief description of each. We use the Sans Serif and the Sm C fonts to refer to the name of a semantic aspect and a semantic option, respectively.

A big step is created by iterating through the stages of the flowchart in Fig. 3a. One iteration of the flowchart corresponds to one small step. For each transition, t , stage 1 determines whether $trig(t)$ is true with respect to the statuses of events, according



(a) Semantic variation points in the operation of a big step

Aspect/Option	Description
Event	<i>Specifies the extent that a generated event is present in a big step:</i>
R	In all snapshots of the big step after it is generated.
N x S S	In the destination snapshot of the small step that it is generated in.
N x B S	In all snapshots of the next big step after the big step it is generated in.
GC Variable	<i>Specifies the snapshot from which values of variables for checking the guard condition of a transition are obtained:</i>
GC B S	Obtains the value of a variable from the beginning of the big step.
GC S S	Obtains the value of a variable from the source snapshot of the small step.
Maximality	<i>Specifies when a sequence of small steps concludes as a big step:</i>
T O	Once a transition, t , is executed during a big step, no other transition whose arena overlaps with the arena of t can be executed.
T M	No constraints on the sequence of small steps.
Concurrency	<i>Specifies the number of transitions that can be taken in a small step:</i>
S	Exactly one transition per small step.
M	More than one transition possible per small step.
Preemption	<i>Specifies whether interrupting transitions can execute in a small step:</i>
P	Two transitions, one an interrupt for the other, cannot be taken together.
N -P	Two transitions, one an interrupt for the other, can be taken together.
Priority	<i>Specifies if a transition has a higher priority than another:</i>
N	For a pair of transitions, t and t' , t is assigned a higher priority than t' by conjoining $\text{trig}(t)$ with the negation of a positive event in $\text{trig}(t')$.
Sc -C	The lower the scope of a transition, the higher its priority is.
Sc -P	The higher the scope of a transition, the higher its priority is.
N P	Neither Sc C nor Sc P is chosen.
RHS Variable	<i>Specifies the snapshot from which values of variables for evaluating the right-hand side (RHS) of an assignment are obtained:</i>
RHS B S	Obtains the value of a variable from the beginning of the big step.
RHS S S	Obtains the value of a variable from the source snapshot of the small step.

(b) Semantic aspects (in Sans Serif font) and their options (in S C font)

Fig. 3. Semantic variations in the operation of a big step

to the **Event** semantic aspect. The three semantic options of this semantic aspect each determines a different extent of a big step (the current big step or the next one) that a generated event is considered as present. Stage 2 determines whether $gc(t)$ is true with respect to the values of variables, according to the **GC Variable** semantic aspect. The first semantic option for this semantic aspect uses the values of variables from the beginning of the big step, whereas the second option uses the up-to-date values of variables from the source snapshot of the small step. Stages 1 and 2, together with the current set of control states of a model, determine the enabledness of individual transitions. Stage 3 checks whether the current big step is **maximal**, according to the **Maximality** semantic aspect, in which case, the big step ends. The first semantic option for this semantic aspect requires that if a transition, t , has been executed in a big step, no other transition whose arena overlaps with t 's can be executed. As an example, according to the first semantic option, in the model in Fig. 1, if t_9 is executed, t_5 cannot be executed because the arena of t_5 overlaps with the arena of t_9 . The second semantic option places no constraint over the maximality of a big step: A big step can continue as long as there are enabled transitions that can be executed.

If the big step is not maximal, stage 4 determines the sets of transitions that can be taken together. Each such set is **complete** in that no transition can be added to it without violating the two related semantic sub-aspects of the **Concurrency** and **Preemption** semantic aspect. The **Concurrency** sub-aspect specifies the number of transitions possible in a small step, with two possible semantic options: one vs. many. When the latter semantic option is chosen, two transitions can be included in the same small step if they are orthogonal. The **Preemption** sub-aspect determines whether a pair of transitions where one is an interrupt for another can be taken together or not. Among the sets of transitions produced by stage 4, the sets that have the highest priority, according to the **Priority** semantic aspect, are chosen by stage 5; the result is the set of potential small steps at the current snapshot. The options for priority semantics include using the negation of events to assign a transition, t , a higher priority than a transition, t' , by including one of the positive events in $trig(t)$ as a negated event in $trig(t')$. For example, in the model in Fig 1, t_3 has a higher priority than t_4 , denoted by $pri(t_3) \succ pri(t_4)$, because $trig(t_3)$ includes *heat* while $trig(t_4)$ includes $\neg heat$. Additionally, if the **N P I IT** semantic option is not chosen, one of the two *hierarchical* priority semantic options is used to compare the priority of two transitions based on their scopes. Stage 6 evaluates the right-hand side (RHS) of an assignment, according to the **RHS Variable** semantic aspect, when executing one of the potential small steps. Similar semantic options as the ones for the **GC Variable** semantic aspect are possible for the **RHS Variable** semantic aspect, but are used to evaluate the RHS of assignments instead of their GCs.

We use the following notation to describe the semantic quality attributes. The **set of enabled transitions** of a model at a snapshot, sp , denoted by $enabled(sp)$, is the set of all transitions such that, for each transition, its source control state is in the current set of control states and it passes stages 1 and 2 of flowchart in Fig. 3a. Similarly, the **set of executable transitions** at snapshot sp , denoted by $executable(sp)$, is the set of all transitions that belong to a potential small step at sp . Formally, $t \in executable(sp) \Leftrightarrow \tau \in potential(sp) \wedge t \in \tau$. By definition, $t \in executable(sp) \Rightarrow t \in enabled(sp)$, but not vice versa, because of the **Maximality** and **Priority** semantic aspects.

In this paper, we consider BSML semantics in which the executability of each small step relies only on its source snapshot [6]. A few of the semantic variations that are not considered here lend themselves to a different set of semantic quality attributes, as will be briefly discussed in Section 4.

3 Semantic Quality Attributes for BSMLs

In this section, we formally describe three semantic quality attributes for BSMLs. For each semantic quality attribute, we enumerate all combinations of the semantic options in Table 3b that each results in a semantics that satisfies the semantic quality attribute.

3.1 Priority Consistency

In a *priority-consistent* BSML semantics, higher priority transitions are chosen to execute over lower priority transitions. A model cannot have two big steps, T_1 and T_2 , where T_1 includes transitions that are all of lower or incomparable priority than T_2 's. A priority semantic option enforces a priority semantics only in the individual small steps of a big step, but not in the whole big step. A priority-consistent BSML semantics is useful since it exempts a modeller from worrying about a high-priority transition being executed in some big steps but not others. We call a semantics that is not priority consistent, *priority inconsistent*.

Example 1. The two models in Fig. 4 are used to demonstrate examples of priority-inconsistent behaviour. Both models are considered when they reside in their default control states, environmental input event i is present, and, for the first model, when $x = 0$. In the model in Fig. 4(a), we consider a BSML semantics that subscribes to the M₋ concurrency semantics, the T_K M₋ maximality semantics, the Sc₋P₋T priority semantics, and the GC SM₋ ST₋ GC variable semantics. Two big steps are possible: $T_1 \rightarrow t_1 t_4 t_2 t_6 \rangle$ and $T_2 \rightarrow t_1 t_4 t_3 t_5 \rangle$. This behaviour is priority inconsistent since $pri(t_6) > pri(t_5)$: the scope of t_6 is the parent of the scope of t_5 , and T_1 executes t_6 but T_2 executes t_5 . For the model in Fig. 4(b), the same semantic options as for the model in Fig. 4(a) are considered, plus the R_MI₋ event semantics. Two big steps are possible: $T'_1 \rightarrow t_1 t_2 t_5 \rangle$ and $T'_2 \rightarrow t_1 t_3 t_4 \rangle$. This behaviour is priority inconsistent since $pri(t_5) > pri(t_4)$ according to the N_T TI₋ semantics: $trig(t_5)$ includes b while $trig(t_4)$ includes $\neg b$, and T'_1 executes t_5 but T'_2 executes t_4 . If the N_T BI₋ ST₋ event semantics had been chosen, which requires a generated event to be present only in the next big step, two priority-consistent big steps would have been possible: $T'_1 \rightarrow t_1 t_2 \rangle$ and $T'_2 \rightarrow t_1 t_3 \rangle$, followed by big steps $t_5 \rangle$ and $t_4 \rangle$, respectively.

Definition 1. A BSML semantics is *priority consistent* if for all BSML models, M ,

$$T_1 T_2 \quad bigsteps(M) \quad (T_1 I \rightarrow T_2 I) \wedge (T_1 sp_0 \rightarrow T_2 sp_0) \quad trans(T_1) \rightarrow trans(T_2)$$

where

$$1 \quad 2 \quad \neg(1 \rightarrow 2) \wedge \neg(2 \rightarrow 1) \quad and$$

$$1 \quad 2 \quad (t_1 \rightarrow t_1 t_2 \rightarrow pri(t_1) \wedge pri(t_2)) \wedge \neg(t_2 \rightarrow t_2 t_1 \rightarrow pri(t_2) \wedge pri(t_1))$$

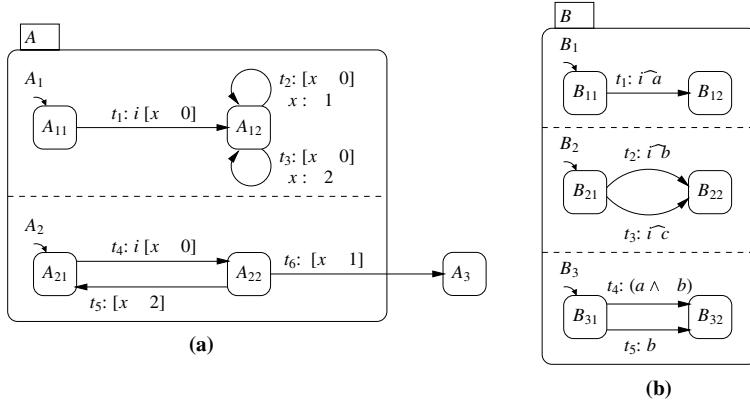


Fig. 4. Examples of priority-inconsistent behaviour

Intuitively, $t_1 \ll t_2$ if it is not the case that t_1 has a transition that has a higher priority than a transition in t_2 without t_2 having such a transition, and also not vice versa.

Priority-Consistent Semantics. Proposition 1 specifies the BSML semantics that are priority consistent. We use the name of a semantic option as a proposition to specify all BSML semantics that subscribe to it.

Proposition 1. A BSML semantics is priority-consistent if and only if its constituent semantic options satisfy predicate $\mathbf{P} = \mathbf{P1} \wedge \mathbf{P2}$, where

$$\mathbf{P1} = \neg N \wedge P_i \wedge IT \wedge T \wedge O \quad \text{and} \quad \mathbf{P2} = N \wedge T \wedge Bi \wedge St$$

Proof Idea. Predicate **P1** is a necessary condition for a BSML semantics to be priority-consistent. This can be proven by contradiction. If **P1** is not true in a priority-consistent BSML semantics, a *counter example model* with a priority-inconsistent behaviour can be constructed. For example, for any BSML that subscribes to the $S_C - P_T$ priority semantics and the $T \wedge M$ maximality semantics, the model in Fig. 5 is such a counter example model: when the model resides in its default control states, two big steps, $T_1 = t_1 \circ t_3$ and $T_2 = t_2 \circ t_2$, are possible, but $trans(T_1) \neq trans(T_2)$ because $pri(t_3) > pri(t_2)$. Predicate **P2** is a necessary condition for a priority-consistent BSML semantics, because for a BSML semantics that subscribes to an event semantics other than the $N \wedge T \wedge Bi \wedge St$ semantics, regardless of its other semantic options, there is a counter example model with a priority-inconsistent behaviour. For example, the model in Fig. 4(b), in Example 1, would have a priority-inconsistent behaviour even if the S_I concurrency and or the $N \wedge T \wedge Sm \wedge St$ event semantic options are chosen. Predicates **P1** and **P2** are also sufficient conditions for priority-consistent BSML semantics, according to a hierarchical and the $N \wedge T \wedge I$ priority semantics, respectively. This can be proven by showing that if **P** is true in a BSML semantics, a high-priority transition is either executable and

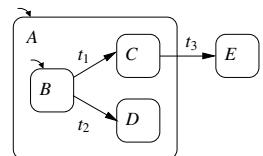


Fig. 5. A counter example

taken during a big step, or it cannot become executable during that big step. If a BSML semantics subscribes to the $S_C - P_T$ ($S_C - C_1$) priority semantics, a transition that has a higher (lower) scope than an already executed transition cannot become executable because of the $T_K O$ maximality semantics. Similarly, for the N_{T_1} priority semantics, if a high-priority transition is not enabled due to the absence of an event, it cannot possibly become enabled in that big step, because the statuses of events do not change. As such, a BSML semantics is priority consistent if and only if P .

3.2 Non-cancelling

In a *non-cancelling* BSML semantics, once a transition of a model becomes executable in a big step, it remains executable during the big step, unless, it is taken by the next small step, it cannot be taken any more because of the maximality constraints, or its scope is the same as or a descendant of the scope of a transition executed in the next small step. A non-cancelling BSML semantics is useful since it exempts a modeller from worrying about an enabled transition of interest mistakenly becoming disabled. We call a BSML semantics that is not non-cancelling, *cancelling*.

Example 2. We consider the model in Fig. 1 when it is in its initial control states and when events *req_u* and *high* are received from the environment. We choose the $T_K M$ maximality semantics. Initially, t_4 is executable. If t_1 is executed in the first small step, making *heat* present and *hot true*, t_4 would not be enabled any more, although the big step is not maximal and the scope of t_4 does not overlap with t_1 's. This is a cancelling behaviour, which can be averted by executing t_1 and t_4 together.

Definition 2. A BSML semantics is non-cancelling if for all BSML models, M ,

$$\begin{aligned} T \text{ bigsteps}(M) \quad i(1 \quad i \quad \text{length}(T)) \quad t \quad \text{executable}(sp_i)_1 \\ (t \quad \tau_i) \vee (t \quad \text{executable}(sp_i)) \vee ((t' \quad \tau_i) \text{ tookone}(t' \quad t) \vee \text{dominated}(t' \quad t)) \end{aligned}$$

where *tookone*($t' \quad t$) is true, if by executing t' , t cannot be taken because of the $T_K O$ maximality semantics of the BSML; and *dominated*($t' \quad t$) is true, if the scope of t is the same as or a descendent of the scope of t' ; e.g., $\text{src}(t) \quad \text{dest}(t')$.

The rationale for including the third disjunct in Definition 2 is twofold; when predicate *tookone*($t' \quad t$) is true, t is not a transition of interest any more; and when predicate *dominated*($t' \quad t$) is true, the execution of t' has entered and or exited the scope of t . As such, whichever predicate is true, it is natural to consider the enabledness of t afresh.

Achieving a non-cancelling BSML semantics not only relies on enabledness and concurrency semantics but also on hierarchical semantics, as the next example shows.

Example 3. We consider a modified version of the model in Fig. 4(a) in which $gc(t_2)$ $gc(t_3)$ true. We consider this new model when it resides in control states A_{12} and A_{22} , and $x = 2$. If a BSML semantics that subscribes to the S_I concurrency, the $S_C - P_T$ priority, and the $GC_Sm - ST_GC$ variable semantic options is chosen, initially both t_5 and t_2 are executable, but if t_2 is executed, t_5 becomes unexecutable because t_6 becomes enabled and $pri(t_6) > pri(t_5)$. If the M concurrency semantics had been chosen, instead of the S_I concurrency semantics, the above cancelling behaviour would have been averted because t_2 and t_5 would have been executed together.

Non-Cancelling Semantics. A non-cancelling semantics can be achieved by one of the following two approaches: (i) once a transition becomes executable, the execution of other transitions does not affect its executability; or (ii) once a transition becomes executable, it is immediately executed, precluding the possibility of becoming unexecutible. For example, by choosing the $N \rightarrow B_1 \rightarrow St$ event semantics together with the $N \rightarrow P_1 \rightarrow IT$ semantics, a non-cancelling semantics via approach (i) can be achieved. By choosing the $N \rightarrow Sm \rightarrow St$ semantics together with a concurrency semantics that ensures that an executable transition is executed immediately, a non-cancelling semantics via approach (ii) can be achieved. Formally,

Proposition 2. *A BSML semantics is non-cancelling if and only if its constituent semantic options satisfy predicate $N \wedge N_1 \wedge N_2 \wedge N_3$, where*

$$\begin{array}{ll} N_1 & (T \rightarrow M \wedge \neg N \rightarrow P_1 \rightarrow IT) \text{ Maximizer} \\ N_2 & N \rightarrow Sm \rightarrow St \text{ Maximizer} \\ N_3 & GC Sm \rightarrow St \text{ Maximizer and} \\ \text{Maximizer } M & \wedge [(T \rightarrow M \wedge N \rightarrow P_1 \rightarrow IT) \rightarrow N \rightarrow P \rightarrow M \rightarrow IT] \end{array}$$

Proof Idea. Predicate N is a sufficient condition for a non-cancelling BSML semantics because it characterizes only non-cancelling BSML semantics. When N_1 , N_2 , and N_3 are all satisfied through their antecedents being false, a non-cancelling semantics according to approach (i) above is achieved, otherwise a non-cancelling semantics according to approach (ii) can be achieved. When the antecedents of N_1 , N_2 , and N_3 are all false, it can be proven that if an executable transition that is not executed in the immediate small step becomes unexecutible, it is because of the maximality semantics, and not because it is not enabled or does not have a high priority any more. When the antecedents of at least one of N_1 , N_2 , or N_3 is true, predicate *Maximizer* requires the M concurrency semantics to ensure that as many as possible enabled transitions are executed immediately. The second conjunct of the *Maximizer* predicate ensures that when the $T \rightarrow M$ semantics is chosen, a pair of executable transitions that have the same priority and one is an interrupt for the other are taken together in the same small step, according to the $N \rightarrow P \rightarrow M \rightarrow IT$ semantics; otherwise, the execution of the lower-scope transition can disable the higher-scope transition. To prove that predicate N is also a necessary condition for any non-cancelling BSML semantics, it can be shown that N characterizes all possible non-cancelling BSML semantics. This can be proven, via proofs by contradiction, to show that: (a) approaches (i) and (ii) above are the only ways to achieve a non-cancelling semantics; and (b) predicate N does not over-constrain the choices of semantic options.

3.3 Determinacy

In a *determinate* BSML semantics, if two big steps of a model in response to the same environmental input execute the same (multi) set of transitions in different orders, their destination snapshots are *equivalent*. An equivalence relation, denoted by “ \sim ”, can be defined with respect to any subset of the snapshot elements. We consider determinacy for both events and variables. A determinate BSML semantics is useful since it exempts a modeller from worrying about the effect of an out-of-order execution of the transitions in a big step. We call a BSML semantics that is not determinate, *non-determinate*.

Definition 3. A BSML semantics is determinate if for all BSML models, M ,

$$\left(\bigcup_{\tau_1 \in \text{smallsteps}(T_1)} \tau_1 \quad \bigcup_{\tau_2 \in \text{smallsteps}(T_2)} \tau_2 \right) \vdash T_1 \text{ sp}_{\text{length}(T_1)} \quad T_2 \text{ sp}_{\text{length}(T_2)}$$

where “ \vdash ” is the sum operator for multisets.

To have determinacy, a BSML must create only single assignment models.

Definition 4. A big step, T , is single assignment if there are no two transitions in the big step that assign values to the same variable. A BSML model, M , is single assignment if all big steps $T \in \text{bigsteps}(M)$ are single assignment.

A crude way to achieve single assignment models is to require the $T \vdash O$ maximality semantics and that at most one transition of a model assigns a value to each variable.

Example 4. The model in Fig. 6 controls the operation of a chemical plant.² The environmental input events *inc_one* and *inc_two* indicate that the amount of a chemical substance in the plant needs to be incremented by one or two units, respectively. We consider the model when: it resides in its default control states, $\text{inc}_1 = \text{inc}_2 = 0$, and the environmental input events *inc_one* and *inc_two* are received together. The model is single-assignment only if environmental input event *reset* is received neither with *inc_one* nor with *inc_two*. If a BSML semantics subscribes to the $M \vdash O$, the $T \vdash M$, the $R \vdash M$, the $GC \vdash ST$, and the $RHS \vdash ST$ semantic options, two big steps are possible: $T_1 \vdash t_1 t_5 t_2 t_7 t_3 t_4$ and $T_2 \vdash t_3 t_5 t_4 t_7 t_1 t_2$. T_1 assigns the value one to *inc* while T_2 assigns value two, which is a non-determinate behaviour because T_1 and T_2 execute the same set of transitions. If the $RHS \vdash ST$ semantic option had been chosen, instead of the $RHS \vdash ST$ semantic option, the assignment to *inc* by t_5 would have read the values of inc_1 and inc_2 from the beginning of the big step, and thus a determinate behaviour would have been achieved.

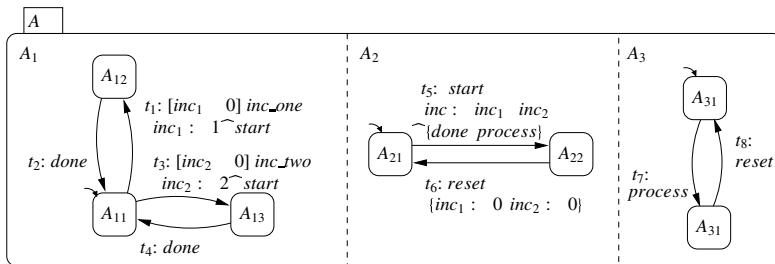


Fig. 6. An example of a non-determinate behaviour

Determinate Semantics. First, we present a lemma that explains why the first semantics in Example 4 is not determinate, but would have been so if the $T \vdash O$ semantic option had been chosen. We then specify the class of all determinate BSML semantics.

² This model is adapted from the sequence-chart model in the motivating example in [1].

Lemma 1. *In a BSML semantics that subscribes to the T κ O and M semantic options, starting from the same snapshot, if two big steps, T_1 and T_2 , of a single-assignment model consist of the same sets of transitions, then they are the same.*

Proof Sketch. This claim can be proven inductively by traversing over the sequence of small steps of T_1 and T_2 . Starting from snapshots $T_1 sp_0$ and $T_2 sp_0$, their first small steps, $T_1 \tau_1$ and $T_2 \tau_1$, should be the same. If not, let us assume that there exists a transition, t , such that $t \in (T_1 \tau_1 \cap T_2 \tau_1)$. However, such a t cannot exist: Transition t can only be not taken by $T_2 \tau_1$ if it is replaced by a $t' \in T_2 \tau_1$ such that t and t' cannot be taken together according to the Concurrency and Preemption semantics. But if that is true, T_2 can never execute t because the T κ O maximality semantics precludes the possibility of such a t being taken after t' had been taken. Thus, it should be the case that $T_1 \tau_1 = T_2 \tau_1$. Similarly, it should be the case that all $T_1 \tau_i$'s and $T_2 \tau_i$'s, $1 \leq i \leq \text{length}(T_1)$, are the same. Therefore, T_1 and T_2 are the same.

Proposition 3. *A BSML semantics is determinate if and only if its constituent semantic options satisfy predicate **D** $\equiv \mathbf{D1} \wedge \mathbf{D2}$, where*

$$\begin{aligned}\mathbf{D1} \quad & \text{RHS Bi St} \vee (\text{RHS Sm St} \wedge (\text{T } \kappa \text{ O } \wedge \text{M })) \text{ and} \\ \mathbf{D2} \quad & (\text{R M I} \vee \text{N T Bi St}) \vee (\text{N T Sm St} \wedge (\text{T } \kappa \text{ O } \wedge \text{M }))\end{aligned}$$

Proof Idea. Predicate **D** is a sufficient condition for a BSML semantics to be determinate. Predicate **D1** deals with determinacy for variables, while predicate **D2** deals with determinacy for events. The first disjunct of **D1** achieves determinacy for variables because of the single-assignment assumption and the fact that early assignments do not affect the later ones. Similarly, the first disjunct of **D2** achieves determinacy for events because the R M I and N T Bi St semantics both accumulate all of the generated events of a big step. The second disjuncts of **D1** and **D2** are valid characterization of determinate BSML semantics because of Lemma 1. Predicate **D** is also a necessary condition for a determinate BSML semantics. If **D1** does not hold for a determinate BSML semantics, then it means that it subscribes to the RHS Sm St assignment semantics but not to both the T κ O maximality semantics and the M concurrency semantics. If the BSML semantics does not subscribe to the T κ O maximality semantics, then the model in Example 4 shows a non-determinate behaviour for such a BSML semantics, which is a contradiction. If the BSML semantics does not subscribe to the M concurrency semantics, but subscribes to the T κ O, a counter example model can be constructed, as follows. We consider the following three orthogonal enabled transitions in a model, $t_1 : a : 1$, $t_2 : b : 1$, and $t_3 : c : a \cdot b$, when $a \cdot b = 0$. Based on the order of the execution of t_1 , t_2 , and t_3 , the value of c is either zero, one, or two, which is a non-determinate behaviour. Similarly, it can be proven that **D2** is a necessary condition for a determinate BSML semantics. Thus, **D** is necessary and sufficient condition for a BSML semantics to be determinate.

4 Related Work

Huizing and Gerth identified the three semantic criteria of *responsiveness*, *modularity*, and *causality* for the S_i concurrency semantics, the T κ O maximality semantics, and events semantics [13] only. Their modularity criterion requires that a generated

event by a model is treated the same as an event received from the environment. The two semantics in their framework that are modular, namely, semantics *A* and *D*, can be shown to be also non-cancelling. Semantics *A* corresponds to the $N \tau Bi St$ event lifeline semantics; semantics *D* corresponds to the W event lifeline semantics. In the W event lifeline semantics, which we informally considered in our deconstruction [7, 8], a generated event is present throughout the big step in which it is generated.

Pnueli and Shalev introduced a *globally consistent* event semantics [18], which is the same as the $R_m i$ semantics except that if the absence of an event has made a transition enabled in a small step, the event is not generated later. Global consistency and priority consistency with respect to the $N \tau_i$ priority semantics are comparable. The difference is that the former is defined at the level of individual big steps whereas the latter is defined at the level of all big steps that have the same source snapshot.

Synchronous languages are used to model program reactive systems that are meant to behave deterministically [9]. We have categorized the un-clocked variations of synchronous languages, such as Esterel [4] and Argos [17], as BSMLs that support the W event lifeline semantics [7, 8]. A model is deterministic if its reaction to an environmental input as a big step always results in a unique destination snapshot. Determinism is related to determinacy: A deterministic semantics is by definition determinate, but not vice versa. A determinate semantics does not preclude the possibility of a model reacting to a single environmental input via two big steps with different sets of transitions. In the presence of variables, determinism can be considered only as a property of a model but not of a semantics, because, as opposed to events, variables can have infinite ranges, making it impossible to handle determinism at the level of the description of a semantics. In the absence of variables, however, deterministic semantics for Esterel [3, 20] and Argos [17] have been developed.

Similar concepts as our semantic quality attributes have been considered in different models of computation, but at the level of models instead of semantics. For example, in Petri nets, the notion of *persistence* [16], which requires a transition to remain enabled until it is taken, is similar to our non-cancelling semantic quality attribute. In asynchronous circuits, the notions of *semi-modularity* and *quasi semi-modularity* are similar to our non-cancelling semantic quality attribute, and the notion of *speed independence* is analogous to our determinacy semantic quality attribute [5, 19]. Janicki and Koutny introduce the notion of *disabling* in the context of a relational model of concurrency [14], which is similar to our priority consistency semantic quality attribute. Lastly, the notions of *persistence* and *determinacy*³ for *program schemata* [15] are analogous to our non-cancelling and determinacy semantic quality attributes, respectively. In general, compared to the aforementioned concepts, (i) our semantic quality attributes are defined for semantics, rather than individual models; and (ii) they are aimed at practical requirements modelling languages, instead of models of computation.

5 Conclusion and Future Work

In this paper, we introduced three semantic quality attributes, namely, non-cancelling, priority consistency, and determinacy, for the family of big-step modelling languages

³ We have adopted the name of our semantic quality attribute from this work.

(BSMLs). When a BSML supports a semantic quality attribute, modellers are exempt from worrying about certain complications of the ordering of transitions in a model. We formally specified the subsets of BSML semantics that satisfy each semantic quality attribute. Next, we plan to identify combinations of meaningful syntactic well-formedness constraints and semantic options that achieve a semantic quality attribute. For example, if the syntax of a BSML is restricted to *simple* transitions, where a transition, t , is simple if $\text{parent}(\text{src}(t)) = \text{parent}(\text{dest}(t))$, then predicate N1 in Proposition 2 can be dropped.

References

1. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. *IEEE Transactions on Software Engineering* 29(7), 623–633 (2003)
2. von der Beeck, M.: A comparison of Statecharts variants. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994 and ProCoS 1994. LNCS, vol. 863, pp. 128–148. Springer, Heidelberg (1994)
3. Berry, G.: The constructive semantics of pure Esterel draft version 3 (1999),

$$\begin{array}{ccccccc} tt & & www & o & & t & o \\ & & & & & & \end{array}$$
4. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming* 19(2), 87–152 (1992)
5. Brzozowski, J.A., Zhang, H.: Delay-insensitivity and semi-modularity. *Formal Methods in System Design* 16(2), 191–218 (2000)
6. Esmaeilsabzali, S., Day, N.A.: Prescriptive semantics for big-step modelling languages. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 158–172. Springer, Heidelberg (2010)
7. Esmaeilsabzali, S., Day, N.A., Atlee, J.M., Niu, J.: Semantic criteria for choosing a language for big-step models. In: RE 2010, pp. 181–190. IEEE Computer Society Press, Los Alamitos (2009)
8. Esmaeilsabzali, S., Day, N.A., Atlee, J.M., Niu, J.: Deconstructing the semantics of big-step modelling languages. *Requirements Engineering* 15(2), 235–265 (2010)
9. Halbwachs, N.: Synchronous Programming of Reactive Systems. Kluwer, Dordrecht (1993)
10. Harel, D.: Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8(3), 231–274 (1987)
11. Heitmeyer, C., Jeffords, R., Labaw, B.: Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology* 5(3), 231–261 (1996)
12. Heninger, K.L., Kallander, J., Parnas, D.L., Shore, J.E.: Software requirements for the A-7E aircraft. Tech. Rep. 3876, United States Naval Research Laboratory (1978)
13. Huizing, C., Gerth, R.: Semantics of reactive systems in abstract time. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, pp. 291–314. Springer, Heidelberg (1992)
14. Janicki, R., Koutny, M.: Structure of concurrency. *Theoretical Computer Science* 112(1), 5–52 (1993)
15. Karp, R.M., Miller, R.E.: Parallel program schemata. *Journal of Computer and System Sciences* 3(2), 147–195 (1969)
16. Landweber, L.H., Robertson, E.L.: Properties of conflict-free and persistent Petri nets. *Journal of the ACM* 25(3), 352–364 (1978)

17. Maraninchi, F., Rémond, Y.: Argos: an automaton-based synchronous language. *Computer Languages* 27(1/3), 61–92 (2001)
18. Pnueli, A., Shalev, M.: What is in a step: On the semantics of statecharts. In: Ito, T., Meyer, A.R. (eds.) *TACS 1991. LNCS*, vol. 526, pp. 244–264. Springer, Heidelberg (1991)
19. Silver, S.J., Brzozowski, J.A.: True concurrency in models of asynchronous circuit behavior. *Formal Methods in System Design* 22(3), 183–203 (2003)
20. Tardieu, O.: A deterministic logical semantics for pure Esterel. *ACM Transactions on Programming Languages and Systems* 29(2), 1–26 (2007)

Formalizing and Operationalizing Industrial Standards*

Dominik Dietrich, Lutz Schröder, and Ewaryst Schulz

DFKI Bremen, Germany

{firstname.lastname}@dfki.de

Abstract. Industrial standards establish technical criteria for various engineering artifacts, materials, or services, with a view to ensuring their functionality, safety, and reliability. We develop a methodology and tools to systematically formalize such standards, in particular their domain specific calculation methods, in order to support the automatic verification of functional properties for concrete physical artifacts. We approach this problem in the setting of the Bremen heterogeneous tool set HETS, which allows for the integrated use of a wide range of generic and custom-made logics. Specifically, we (i) design a domain specific language for the formalization of industrial standards; (ii) formulate a semantics of this language in terms of a translation into the higher-order specification language HASCASL, and (iii) integrate computer algebra systems (CAS) with the HETS framework via a generic CAS-Interface in order to execute explicit and implicit calculations specified in the standard. This enables a wide variety of added-value services based on formal reasoning, including verification of parameterized designs and simplification of standards for particular configurations. We illustrate our approach using the European standard EN 1591, which concerns calculation methods for gasketed flange connections that assure the impermeability and mechanical strength of the flange-bolt-gasket system.

1 Introduction

Industrial standards are documents that establish uniform engineering or technical criteria of an item, material, component, system, or service, and are designed to ensure its safety and reliability. To that end, they introduce a precise nomenclature for a limited domain and provide an explicit set of requirements that the item at hand has to satisfy, together with methods to check these requirements. E.g., the European standard EN 1591 [18] defines design rules for gasketed flange connections (see Fig. 1) that guarantee impermeability and mechanical strength of the flange-bolt-gasket system in the form of numerical constraints as well as explicit calculation methods to compute or approximate physical quantities. Performing these calculations in order to verify the target properties for a concrete object can be highly time-consuming and costly; hence, several ad hoc software solutions have been developed to support such calculations (e.g., [4]).

We develop a methodology and tools to systematically formalize such standards, in particular their domain specific computation strategies, to support the automatic verification of the requirements for a concrete physical object. We are particularly interested

* Work performed as part of the project FormalSafe funded by the German Federal Ministry of Education and Research (FKZ 01IW07002).

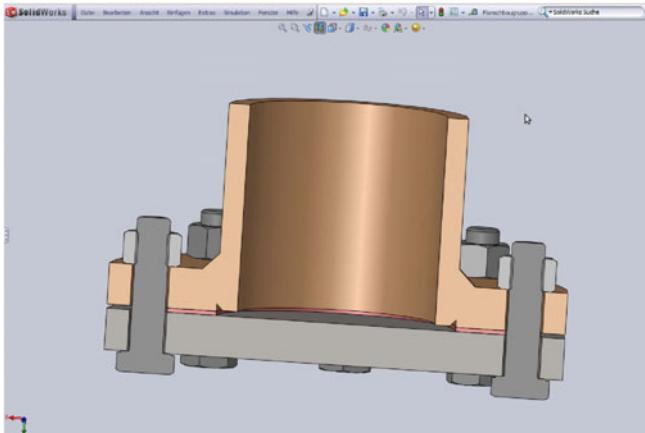


Fig. 1. A flange-bolt-gasket system

in standards giving a guarantee for some functional properties of the system by providing a calculation method. It suffices then to satisfy the criteria of this method in order to ensure these functional properties.

Instead of developing a new system from scratch, our approach consists of designing a new specification language for industrial standards and embedding this language in the institution-based heterogeneous tool integration framework HETS [14], which provides sound and general semantic principles for the integration and translation between different specification languages/logics. This not only allows the reuse of the existing generic machinery provided by the HETS framework, but also gives us direct access to all systems that have already been integrated into the framework, such as the theorem prover Isabelle [15].

The structure of this paper is as follows: Sec. 2 introduces the standard EN 1591. Sec. 3 gives a short overview of relevant parts of the HETS system. The specification language EnCL used to formalize industrial standards is described in Sec. 4 where we also illustrate the architecture of our verification framework for EnCL specifications. We conclude the paper in Sec. 5.

Related work. While there is scattered work on ontological approaches to engineering artifacts, in particular CAD objects (e.g., [1,6]), there is to the best of our knowledge only little existing work on actually formalizing the content of industrial standards, in particular as regards calculation methods. In [12], a more global viewpoint on our overall research goals is given, while the technical results are more oriented towards the representation of CAD geometry. A knowledge-based approach ensuring the safety of pressure equipment is presented in [5]; the formalization in this approach requires more effort than in our framework due to the granularity of the ontology in question. Our approach to formalizing calculations in a logical framework is to some extent related to biform theories [7], the main differences being that we refrain from explicit manipulation of syntax (which instead is left to the CAS operating in the background)

$$W_F = (\pi/4) \times \{f_F \times 2 \times b_F \times e_F^2 \times (1 + 2 \times \Psi_{opt} \times \Psi_Z - \Psi_Z^2) + f_E \times d_E \times e_D^2 \times c_M \times j_M \times k_M\}$$

$$e_D = e_1 \times \left\{ 1 + \frac{(\beta - 1) \times l_H}{\sqrt[4]{(\beta/3)^4 \times (d_1 \times e_1)^2 + l_H^4}} \right\}$$

$$f_E = \min(f_F; f_S)$$

$$\delta_Q = P \times d_E / (f_E \times 2 \times e_D \times \cos\phi_S); \quad \delta_R = F_R / (f_E \times \pi \times d_E \times e_D \times \cos\phi_S)$$

Fig. 2. Some typical definitions from the EN 1591

and moreover work with a loose coupling of algorithmic and axiomatic content via the HETS framework, rather than join the two types of information in mixed theories.

2 Industrial Standards

We proceed to give an overview of the calculation method of the European standard EN 1591. A calculation method is a set of equations, constraints, value tables and instructions. These instructions comprise iterative approximations of a given quantity up to a target accuracy and specify the computation of all data relevant for the checking of the constraints from a set of initial data, which needs to be provided by the user. Figure 3 gives a rough overview of the control flow of the calculation method with an inner and an outer loop. Figure 4 gives a glimpse of how the corresponding instructions look like. It is rather uncommon in mechanical engineering to specify information other than mathematical formulas such as can easily represent this instruction.

Further notable features of the calculational structure of the standard are:

- Most definitions of constants in the standard are given explicitly by equations (see Fig. 2) and in rare cases by value tables corresponding to conditional definitions.

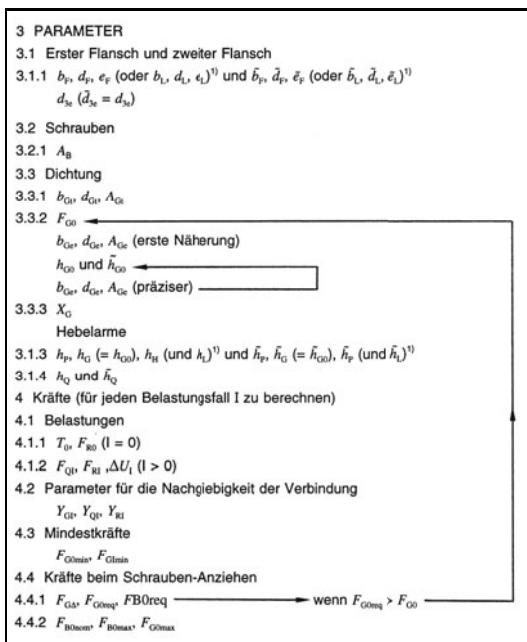


Fig. 3. A control flow diagram from the EN 1591

than mathematical formulas such as equations and inequalities in a formal way, but we can easily represent this instruction as a repeat-until command.

- Constants are sometimes used in the document before they are defined.
- Constants are interpreted as real numbers, and the defining terms contain real constants (π), roots ($\sqrt{\cdot}$), elementary functions such as \sin and \cos , and other functions, e.g., absolute value, \min and \max , to name only a few.
- The standard contains implicit definitions which can be expressed as optimization problems. This renders the evaluation of the calculation method more challenging.

Effective width of gasket:

$$b_{Ge} = \min(b_{Gi}, b_{Gt}) \quad (38)$$

[...] First approximation: $b_{Gi} = b_{Gt}$

More specific value:

$$b_{Gi} = \sqrt{\phi(h_{G0})} \quad (40)$$

$$h_{G0} = \psi(b_{Ge}) \quad (41)$$

[...] Continue evaluating equations (38) to (41) until the value of b_{Ge} does not change any more w.r.t. a precision of 0.1%.

Fig. 4. Iterative Approximation in the EN 1591

3 The Heterogeneous Tool Set

The strategy we pursue to provide formal modelling support for industrial standards and more generally for numerical calculations in engineering is to embed a suitable domain-specific language, to be described in Sec. 4, into the *Bremen Heterogeneous Tool Set* (HETS) [14], which integrates a wide range of logics, programming languages, and tools that enable formal reasoning at various levels of granularity. This includes, e.g., ontology languages, equipped with efficient decision procedures, and first-order languages, for which some degree of automated proof support is available, as well as higher-order languages and interactive provers. HETS follows a multi-lateral philosophy where logics and tools are connected via a network of translations, instead of embedding everything into a central interchange logic. The unifying semantic bracket underlying these translation mechanisms and acting as an abstraction barrier in the implementation framework is the theory of *institutions*; we shall not repeat the formal definitions of the concepts of this theory here, but will provide some intuitions concerning its core points.

Our approach will specifically focus on three items in the HETS network:

- the above-mentioned domain-specific language for engineering calculations, EnCL (*Engineering Calculation Language*), which has an intermediate character between a logic and programming language;
- a computer algebra system (CAS) as an execution engine for EnCL; and
- the higher-order wide-spectrum language HASCASL [16], which serves to make the formal semantics of calculations explicit and to enable advanced reasoning on standards, calculations and concrete engineering designs.

The heterogeneous logical approach involves various translations between these nodes, in particular the following:

- a translation (a so-called *theoretical comorphism*, a concept discussed in some more detail below) from EnCL to HASCASL which identifies EnCL as sublogic of HASCASL;
- a converse translation from the relevant sublogic of HASCASL to EnCL, which enables the use of the CAS as a lemma oracle for HASCASL; and
- the translation from EnCL into the input language of the CAS.

Fig. 5 shows the structure of the HETS logic graph including the above-mentioned features. In the present work, we focus on using the last translation, whose implementation in HETS is already available; implementation of the other two translations is under way.

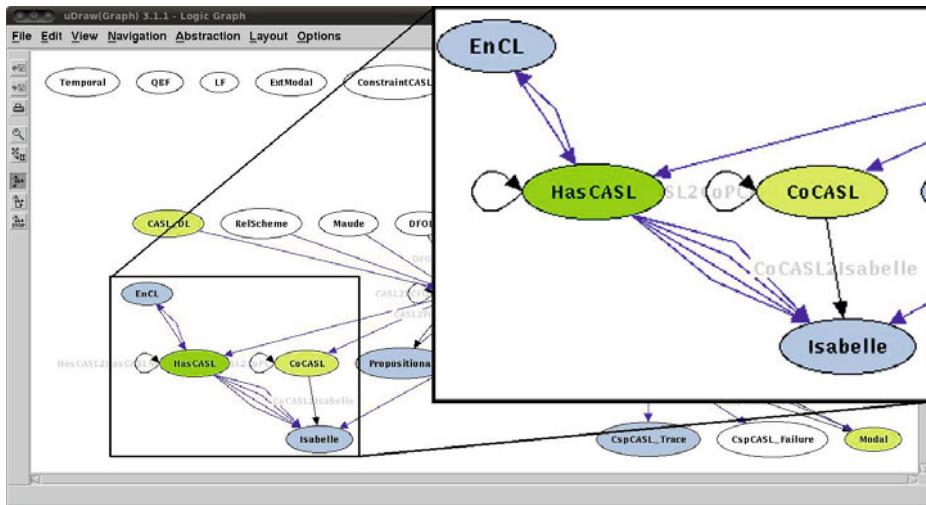


Fig. 5. The HETS logic graph

We now give a brief and informal overview over the concepts of institution and institution comorphism to pave the ground for the description of the respective features in the following sections.

To begin, an *institution* is an abstract logic, construed in a broad sense. It consists of

- a category of *signatures* Σ and *signature morphisms* $\sigma : \Sigma_1 \rightarrow \Sigma_2$ that specify languages of individual theories in the given logic, typically to be thought of as consisting of structured collections of symbols, and their translations, typically renamings and extensions of the set of symbols;
- for each signature Σ , a class of Σ -*models* interpreting its symbols and a set of Σ -*sentences* formed using these symbols, together with a *satisfaction relation* \models between models M and sentences ϕ ;
- for each signature morphism $\sigma : \Sigma_1 \rightarrow \Sigma_2$, a *sentence translation map* σ that applies the corresponding symbol renaming to Σ_1 -sentences and a *model reduction*

that reduces Σ_2 -models M to Σ_1 -models $M|_{\sigma}$, typically by interpreting a Σ_1 -symbol s in the same way as M interprets $\sigma(s)$.

Moreover, one requires the *satisfaction condition* to hold which essentially states that satisfaction is invariant under signature morphisms, i.e. $M|_{\sigma} \models \phi$ iff $M \models \sigma(\phi)$.

One of the generic notions built on top of this structure is that of a *theory* $\mathcal{T} = (\Sigma, \Phi)$ consisting of a signature Σ and a set Φ of Σ -formulas. The formulas in Φ are standardly regarded as *axioms*. HETS offers the additional facility to mark formulas in the theory as *implied*, i.e. as logical consequences of the axioms; this gives rise to proof obligations which can be discharged using proof tools associated to the current logic node. A further source of proof obligations are *theory morphisms*, specified as *views* in HETS; here, a theory morphism $(\Sigma_1, \Phi_1) \rightarrow (\Sigma_2, \Phi_2)$ is a signature morphism $\sigma : \Sigma_1 \rightarrow \Sigma_2$ that transforms all axioms of Φ_1 into logical consequences of Φ_2 .

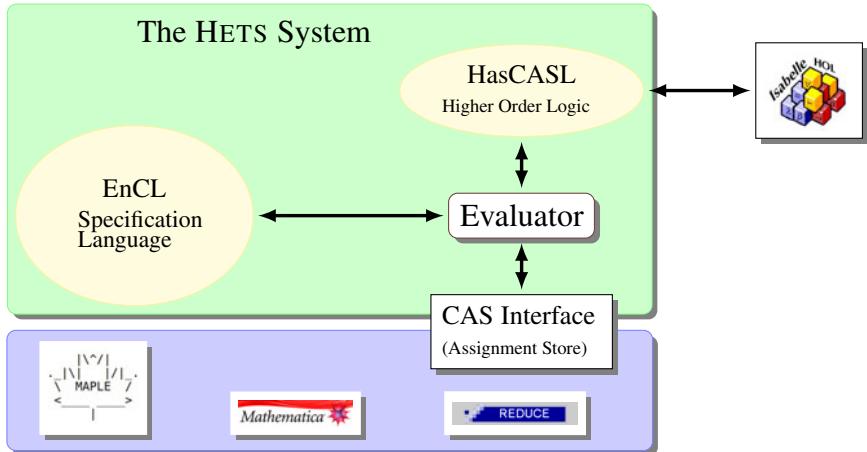


Fig. 6. Architecture of the EnCL Framework

A further logic-independent feature is the ability to build specifications in a modular way by naming, translating, combining, parameterizing and instantiating specifications. These mechanisms are collectively referred to by the term *structured specification*. We will briefly explain some of these mechanisms when they appear in our examples later.

The primary mechanisms used to relate institutions in HETS are *comorphisms* which formalize the notion of logic translation. A comorphism between institutions I and J consists of

- a translation Φ of signatures in I to signatures in J ; and
- for each signature Σ in I , a translation α of Σ -sentences into $\Phi(\Sigma)$ -sentences and a reduction β of $\Phi(\Sigma)$ -models to Σ -models,

again subject to a *satisfaction condition*, in this case that $\beta(M) \models \phi$ iff $M \models \alpha(\phi)$. Below, we will explain how EnCL is cast as an institution, and hence integrated as a logic node in HETS, and how the various translations work.

4 A Domain-Specific Language for Engineering Calculations

Next, we describe the specification language EnCL, which is primarily designed to represent the calculation method of industrial standards but can be used for engineering calculations in general. Our design goals for EnCL are (1) staying close to the informal original description of the standard to minimize the formalization effort and to be readable for engineers and (2) having a precise semantics which allow the formal treatment of the specifications particularly formal verification. Before going into the details we describe briefly the architecture of the EnCL framework for operationalizing industrial standards. EnCL specifications are formalizations of a calculation method, which, given an appropriate set of input values, can be executed. This happens through an evaluator component (see Fig. 6) which communicates with a computer algebra system to evaluate EnCL terms and generates verification conditions that are sent to formal proof assistants for proving.

$<\text{Assignment}>$	$::= \mathbf{c} := <\text{Term}> \mid \mathbf{f}(x_1, \dots, x_n) := <\text{Term}>$
$<\text{Case}>$	$::= \mathbf{case} <\text{Boolterm}> : <\text{Program}>$
$<\text{Cases}>$	$::= <\text{Case}>^+ \mathbf{end}$
$<\text{Sequence}>$	$::= \mathbf{sequence} <\text{Program}> \mathbf{end}$
$<\text{Loop}>$	$::= \mathbf{repeat} <\text{Program}> \mathbf{until} <\text{Boolterm}>$
$<\text{Command}>$	$::= <\text{Assignment}> \mid <\text{Cases}> \mid <\text{Sequence}> \mid <\text{Loop}>$
$<\text{Program}>$	$::= <\text{Command}>^+$
$<\text{Range}>$	$::= \mathbf{vars} \mathbf{v} \mathbf{in} <\text{Set}>$
$<\text{Spec}>$	$::= (<\text{Command}> \mid <\text{Range}>)^+$

Fig. 7. Syntax of EnCL

4.1 Syntax

The EnCL language consists of two layers, one for the definition of the constants used in the calculation and the other for the specification of the calculation method itself. Fig. 7 gives an extended BNF-like grammar of the EnCL language omitting the details of the very standard term-language which is inspired by input languages of computer algebra systems such as Reduce [10], Maple [13] and Mathematica¹. Following our observations of Sec. 2 the language provides simple assignments, conditional statements and unbound conditional iteration.

Given a signature Σ of predefined constants, functions (see Sec. 2 for some examples) and predicates (essentially comparison operators) we define the following language constructs:

Terms are built as usual over Σ and user defined symbols, i.e., constants and functions. We support also special functions which are binders such as $\text{maximize}(t, x)$ defined as the value x' maximizing the term t considered as a function depending on the variable x . This is because we do not support lambda abstraction as this concept is mostly

¹ <http://reference.wolfram.com/mathematica/guide/Mathematica.html>

unused in the mechanical engineering community. We also make a difference between Boolean valued terms and numerical terms and do not allow constants to be assigned to a Boolean value because we do not need it in our current setting.

Assignments of user defined symbols, which are constants c or function patterns $f(x_1, \dots, x_n)$, to terms t .

Conditionals. We support conditionals on the command-level, i.e., not inside terms. We will distinguish later between conditionals in programs and conditional assignments which contain only assignments after being flattened, i.e., conditionals which contain only conditional assignments and assignments.

Sequences allow the marking of a sequence of commands, typically assignments, explicitly as a program sequence instead of treating them as assignments. This is important for the evaluation of a specification as described in the next section.

Loops may contain the convergence predicate in the exit condition beside the predefined comparison operators. Let t be a term which is meant to converge inside a repeat loop and e an expression denoting the acceptable tolerance of t .

convergence(e, t) is defined to be true if and only if the difference of the value of t before the current loop evaluation and afterwards is in the interval $[-|e|, |e|]$.

Fig. 8 shows an excerpt of the specification of the EN 1591. The assignments from Fig. 2 are mainly kept unchanged with the exception that W_F is represented as a function with two arguments. This was necessary because further in the standard W_F is really used as a function, i.e., the arguments k_M and Ψ_Z are used as variables and not as constants.

```

library ENCL/EN1591
logic ENCL
spec EN1591[FLANGEPARAMETER] = ...
  W_F(k_M', Psi_Z') := Pi / 4 * (f_F * 2 * b_F * e_F ^ 2
    * (1 - Psi_Z' ^ 2 + 2 * Psi_opt * Psi_Z')
    + f_E * d_E * e_D ^ 2 * c_M * j_M * k_M') % (eq74)%
  e_D := e_I * (I + (beta - I) * l_H / fthrt((beta / 3) ^ 4
    * (d_I * e_I) ^ 2 + l_H ^ 4)) % (eq75)%
  f_E := min(f_F, f_S) % (eq76)%
  delta_Q := P * d_E / f_E * 2 * e_D * cos(phi_S) % (eq77.1)%
  delta_R := F_R / f_E * Pi * d_E * e_D * cos(phi_S) % (eq77.2)%

```

Fig. 8. EnCL specification of the standard EN 1591

4.2 Semantics

There are two notions of semantics to be distinguished here. First, we want to represent the background theory of the objects dealt with in EnCL specifications, i.e., real numbers and real functions, and second we want to give a meaning to a EnCL specification as a whole in order to talk about the execution and the correctness of a EnCL specification.

Theory of Real Functions

The reason why we need a formalization of the theory of real functions is that we want to prove the correctness of computations which are specified in a EnCL specification and carried out by the evaluator component of our framework. The expressions in the computations refer to elementary real functions such as \cos ; hence we need to do the proofs in the context of a theory specifying those functions. Rather than formalize the required portion of analysis from scratch, which is itself a time-consuming endeavor [3], we base our approach on formal tools that provide a library containing an appropriate theory in our case Isabelle/HOL [15] and MetiTarski [2]. We focus on the Isabelle/HOL system because HETS already provides an interface to Isabelle and the interactive setting in Isabelle seems better suited for first experiments, in particular the Approximation theory² developed by Hoelzl providing proof support for inequalities over the reals [11]. In a later stage of the project where we aim at full automation of the correctness proofs, however, we plan to integrate MetiTarski. Both systems formalize elementary functions such as sine and cosine by Taylor series (e.g. in Isabelle/HOL in the Transsscedental theory³).

Evaluation

The EnCL specification language contains as sub-language the language of assignments, i.e., specifications which consist only of assignments and conditional assignments as defined in Sec. 4.1. We will call commands of this sub-language simply assignments. A specification can hence be divided into a program skeleton containing only non-assignments, e.g., repeat-loops and sequences, and assignments. We require that for the assignments (1) there is at most one assignment for each constant and (2) there is no cyclic dependency between two constants, i.e., the dependency graph for the assignments has no cycles. We split the evaluation according to the divided specification into the evaluation of the program skeleton and an assignment store. This assignment store supports requests to evaluate a term containing constants which are defined by some assignments in the store. The result is a fully evaluated expression, where all defined constants were recursively substituted by their assigned value in the current environment. Typically, a specification splits into a small program containing only a few assignments (see Fig. 9 for an example) and a big assignment store. Fortunately many computer algebra systems support exactly the feature of full evaluation required from an assignment store and can hence be used as such in our framework. Currently we support the computer algebra systems Reduce, Maple and Mathematica.

Verification

Within our framework, the evaluator component interprets EnCL programs and uses an external assignment store with which it communicates via a generic interface, the CAS-Interface. The complicated parts of the evaluation, namely the evaluation and computation of the terms, are outsourced to the CAS, and are in general not guaranteed to be carried out correctly. There are many examples for erroneous computations in CAS [11]

² http://isabelle.in.tum.de/dist/library/HOL/Decision_Procs/Approximation.html

³ <http://isabelle.in.tum.de/dist/library/HOL/Transcendental.html>

```

spec EN1591[FLANGEPARAMETER] = ...
  repeat
    . repeat
      . b_Gi := sqrt(e_G / Pi * d_Ge * E_Gm
        / (h_G0 * Z_F / E_F0 + h_G0 * Z_F / E_F0)
        + (F_G / Pi * d_Ge * Q_maxy) ^ 2)
    until convergence(1.0e-3, b_Ge)
until convergence(1.0e-3, F_G) and F_G0req <= F_G % (eq54)%

```

Fig. 9. EnCL program skeleton of the standard EN 1591

which justify our prudence in this respect even if most of the computations give correct results.

We verify a computation as follows (see Sec. 4.4 for an example). Before we start the evaluation we mark all constants in the assignment store. A *marked constant* stands for the fact that its value has *changed*. We have two rules for marking and unmarking constants:

1. When the evaluator is at the position of an assignment, all constants affected by this assignment are marked.
2. When we generate a verification condition for an assignment, the assigned constant is unmarked.

We trigger the generation of verification conditions when the evaluator is at one of the following three positions: (1) at an assignment, (2) at an until condition of a repeat loop or (3) at a case condition. Depending on the position of the evaluator we generate a verification condition for the assignment in the first case and for the condition in the other cases. In addition we generate in all three cases verification conditions for the assignments of all marked constants which affect the value of the expression in question, namely the assignment in case (1) and the condition in case (2) and (3). This method guarantees that we do not produce obvious copies of already generated verification conditions. For verification condition generation purposes we can treat a condition Φ which is a Boolean term exactly as we treat assignments: we consider Φ as the assignment $b := \Phi$ with an auxiliary constant b , therefore we will only describe the generation of verification conditions for the assignment case. Given an assignment $y := t(x_1, \dots, x_n)$ depending on the constants x_1, \dots, x_n we generate the verification condition as follows. For each x_i we request its value v_i from the assignment store. We then request the value w for the expression $t(x_1, \dots, x_n)$. The condition that the result is correct w.r.t. the current environment is now expressed as the equation $t(v_1, \dots, v_n) = w$. If we can prove this equation in the context of the background theory, then we have proved the correctness of the computation of y .

In addition to the verification of isolated assignments we want to formally specify the evaluation semantics described in the previous paragraph, in order to support the full formal verification of a run of the calculation method. For this purpose we specify the semantics in HASCASL and keep it parametric over the background theory of the term language. This allows us to test different prover back-ends in parallel with different instantiations of the background theory.

Numerical Expressions and Uncertainty Propagation

An important issue for the CAS-Interface are numbers with a bounded precision. Consider, e.g., that we want to accept a certain tolerance in the input values and we provide as input to the computation instead of a value an interval, representing the bounds for the value. On the other hand the assignment store could also be limited concerning the precision of the result and give us only a numerical approximation instead of an exact representation of the result. In order to treat such imprecise values correctly we need the assignment store to support uncertainty propagation. Current computer algebra systems provide here only limited support. Mathematica uses significance arithmetic as built-in support for uncertainty propagation [17] whereas the *intpakX* package [9] provides support for interval arithmetic in Maple which unfortunately does not apply to computations from other packages such as *Optimization*.

In the presence of uncertain values we have to review the generation of verification conditions. An uncertain value v in the assignment store is an interval, i.e., lower and upper bound ($v = [\underline{v}, \bar{v}]$) for the actual value, and we have to generate instead of an equation two inequalities with a common precondition expressing the bounds for the input values. Replacing the inequalities by interval-membership, the verification condition from the previous paragraph becomes

$$\forall x_1, \dots, x_n. (\bigwedge_{i=1}^n x_i \in v_i) \Rightarrow t(x_1, \dots, x_n) \in w$$

4.3 EnCL as an Institution

As indicated in Sec. 3, we need to define an institution for EnCL in order to integrate EnCL into the HETS network. This is not an entirely straightforward enterprise as EnCL mixes logical features with traits that are more typical of a programming language. A definition which leads to a relatively smooth integration with the existing logic graph is the following.

- *Signatures* in EnCL are just collections of function symbols with associated arities, including nullary functions, i.e. constants. These are understood as functions on the space of real numbers.
- A *model* of a signature is just an environment, i.e. an assignment of an n -ary function on the reals to every n -ary function symbol in the signature. Intuitively (although not formally), there is the slight twist here that environments are variable: executing a program will typically modify the environment.
- EnCL has two types of *sentences*:
 - The syntax described in Sec. 4.1 yields *programs*, which form one type of EnCL-sentences.
 - A second type of sentences called *answers* captures the results of actually running a EnCL program. These sentences are not typically expected to be input by the user (although this is technically possible, e.g., in order to simplify the answers manually), but are instead generated as lemmas after running a EnCL program using the back-end CAS. The syntactic format for answers are pairs (p, η) consisting of a program p and an environment η , to be understood intuitively as the statement ‘correct evaluation of p yields the environment η ’.

- Corresponding to the two types of sentences, there are two cases in the definition of satisfaction:
 - A model *satisfies* a *program* p if p terminates successfully when run in the corresponding environment (e.g., when the desired factorizations or minima exist).
 - A model η *satisfies* an *answer* (ϕ, η') if correct execution of p in the environment η terminates successfully and yields η' .

This definition is designed in such a way that the translation of calculation goals in EnCL programs (such as minimization of a function) into existential formulas is sound. Formally, we have the following comorphism from a sublogic of HASCASL to EnCL, which serves the purpose of making EnCL available as a calculational tool in specification development. To begin, the relevant sublogic of HASCASL is the one given by all theories that

- extend the standard HASCASL theory of the real numbers;
- introduce no additional signature except constants of types that occur as input or result types in EnCL statements, i.e. n -ary real functions for $n \geq 0$;
- introduce formulas only in a limited syntax mirroring the abilities of EnCL. Specifically, sentences can be of the form (1) $\exists x. \phi(x)$, or (2) $\phi(c)$, where ϕ corresponds to the semantics of a CAS calculation and, e.g., states that x is the set of zeros of a given polynomial, the minimum of a given function, the factorization of a given number etc., and c is a constant. Moreover, all formulas are marked as implied, i.e. no new axioms are introduced. Formulas of type (1) represent goals, and as such are meant to be input by the user, while formulas of type (2) represent answers from the CAS, typically generated automatically as exported lemmas as described in Sec. 4.2.

Theories in this sublogic are translated into EnCL along a comorphism which suppresses the explicit theory of the reals (which is implicit in EnCL) and otherwise behaves as follows.

- Signatures remain unchanged, except that the explicit type information present in the original HASCASL signature is erased.
- Existential formulas of the form (1) are replaced by assignments $x := e$ where e is the CAS statement corresponding to ϕ , e.g., a factorization or minimization statement. Formulas of the form (2) are trivially reinterpreted as answers in the sense of the definition of EnCL sentences.
- Model reduction trivially reinterprets EnCL Models as HASCASL models.

It is easy to check that this does indeed constitute a comorphism, i.e. fulfills the satisfaction condition. Note that this is independent from the fact that the CAS itself may be buggy – in a manner of speaking, EnCL is an abstraction of the workings of the CAS which presupposes correctness.

Remark 1. One could extend the definition of the relevant sublogic of HASCASL so as to exploit the full programming power of EnCL. As this does not really yield additional insights conceptually, details are omitted.

Conversely, we have a translation of EnCL into HASCASL which makes the semantics of EnCL programs explicit and hence enables full formal reasoning over entire calculations. It takes the shape of a so-called *theoroidal comorphism* where we associate to each EnCL signature not just a HASCASL signature (as in a plain comorphism) but a HASCASL theory, which imports a HASCASL specification of the EnCL semantics. The former is just a straightforward encoding of the transformations on environments effected by the various EnCL constructs. A EnCL program then induces a partial function p representing its semantics, and as a EnCL sentence is translated into a definedness assertion amounting to the statement that p terminates when run from the specified state. A EnCL answer (p, η) is just reinterpreted as the obvious formula stating that running p yields η . As indicated above, this translation enables fine-grained reasoning over EnCL calculations, including, besides the verification of individual results of the CAS, the verification of entire EnCL programs.

```

spec POLYFACTOR =
  .  $z0 := z4 + z3 + 20 \%(\text{coef}0)\%$ 
  .  $z2 := 3 * z3 - 30 \%(\text{coef}2)\%$ 
  .  $z4 := 15 \%(\text{coef}4)\%$ 
  sequence
    .  $z := \text{factor}(x^5 - z4 * x^4 + z3 * x^3 - z2 * x^2 + z1 * x - z0)$ 
  end
end

```

$\%(\text{program})\%$

Fig. 10. EnCL specification for polynomial factorization

4.4 Example

To illustrate the work-flow of a calculation in the EnCL framework, we process the specification shown in Fig. 10 step-by-step. The static analysis splits the specification into an assignment store with five assignments, $\text{coef}0, \dots, \text{coef}4$, and a program consisting of the single assignment in the sequence block. The evaluator moves the assignment store to a computer algebra system via the CAS-Interface and puts the instruction pointer at the single assignment of the program. At the beginning all assignments in the assignment store are marked. The assignment for z depends on the constants x and $z0$ to $z4$, but there are only assignments for $z0$ to $z4$ in the assignment store from which we can generate verification conditions. Hence we order the corresponding assignments w.r.t. the dependency graph and generate the verification conditions for them. This produces only trivial verification conditions such as $15 + 85 + 20 = 120$. The constants $z0$ to $z4$ are now unmarked. In the next step, the evaluator stores the current assignment in the assignment store and generates the verification condition for this assignment,

$$\begin{aligned} & \text{factor}(x^5 - 15 * x^4 + 85 * x^3 - 225 * x^2 + 274 * x - 120) \\ &= (x - 5) * (x - 4) * (x - 3) * (x - 2) * (x - 1) \end{aligned}$$

This verification condition is translated to Isabelle and proved by a short Isar-proof in three steps Fig. 11. The background theory is based on the Isabelle formalization of the reals where we added a definition for the (nearly dummy) factor operator requiring only that factorizing a term does not change its value.

```

theory factor imports Real
begin
constdefs factor :: "real => real" "factor(x) == x"
theorem factor1 : "!! x. factor(x^5-15*x^4+85*x^3-225*x^2+
274*x-120) = (x-5)*(x-4)*(x-3)*(x-2)*(x-1)"
(is "!! x. factor (?a x) = (?b x)")
proof -
fix x::real
have "(x-5)*(x-4)*(x-3)*(x-2)*(x-1) =
x*x*x*x*x - (x*x*x*x)*15 + (x*x*x*x)*85 - (x*x*x*x)*225 + 274*x - 120"
by (simp add: ring_simps)
also have "... = x^5-15*x^4+85*x^3-225*x^2+274*x-120"
by (simp add: Groebner_Basis.class_semiring.semiring_rules)
also have "... = factor (?a x)" by (simp add: factor_def)
finally show "factor (?a x) = (?b x)" by simp
qed

```

Fig. 11. Isabelle proof for the validity of the polynomial factorization

5 Conclusion

We have developed a methodology to formalize industrial standards and a method to execute such formalizations based on the HETS framework. Specifically, we have designed a domain specific language EnCL for engineering calculations which allows for the formulation of a given calculation method that stays close to the original formulation in the standard. The integration of this language into the heterogeneous logic framework HETS enables us to relate these specifications to theories available in HAS-CASL, such as ontological summaries of CAD designs [8] or abstract geometric representations of CAD objects [12], in order to automate the parameter extraction for the concrete computation. We have also integrated a computer algebra system (CAS) interface into HETS and instantiated it with several state-of-the-art CAS. This allows us to outsource the calculational part of EnCL specifications, which is a rather natural choice to handle the computations in the presence of implicit definitions, such as references to the argument value which minimizes a function in a given range. A key point here was to cast a mainly procedural input language for a CAS as an institution.

Potential benefits of the formal approach beyond the applications presented here include

- statement and proof of formal consequences of the prescriptions of the standard, e.g., explicit formulas for maximum calculations;
- partial instantiations of the standard to particular situations and ensuing simplification of the calculation procedures (e.g., when some parameters become 0, a fairly typical situation);
- full formal verification of designs.

Acknowledgements

The work reported here was supported by the FormalSafe project conducted by DFKI Bremen and funded by the German Federal Ministry of Education and Research (FKZ 01IW07002). We gratefully acknowledge useful discussions with Till Mossakowski.

References

1. Abdul-Ghafour, S., Ghodous, P., Shariat, B., Perna, E.: A common design-features ontology for product data semantics interoperability. In: IEEE/WIC/ACM International Conference on Web Intelligence, WI 2007, pp. 443–446. IEEE Computer Society, Los Alamitos (2007)
2. Akbarpour, B., Paulson, L.C.: Metitarski: An automatic theorem prover for real-valued special functions. *J. Autom. Reasoning* 44(3), 175–205 (2010)
3. Barendregt, H., Wiedijk, F.: The challenge of computer mathematics. *Phil. Trans. R. Soc. A* 363(1835), 2351–2375 (2005)
4. Bullack, H.-J.: Flanschberechnungen nach EN 1591. Kamprath interaktiv, 1st edn. (2006)
5. Camossi, E., Giannini, F., Monti, M., Brogotto, P., Pittiglio, P., Ansaldi, S.: Ontology Driven Certification of Pressure Equipments. *Process safety progress* 27(4), 313–322 (2008)
6. Colombo, G., Mosca, A., Sartori, F.: Towards the design of intelligent cad systems: An ontological approach. *Advanced Engineering Informatics* 21(2), 153–168 (2007)
7. Farmer, W.M.: Biform theories in chiron. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM/CALCULEMUS 2007. LNCS (LNAI), vol. 4573, pp. 66–79. Springer, Heidelberg (2007)
8. Franke, M., Klein, P., Schröder, L.: Ontological semantics of standards and plm repositories in the product development phase. In: Proc. 20th CIRP Design Conference 2010. Springer, Heidelberg (to appear, 2011)
9. Grimmer, M., Petras, K., Revol, N.: Multiple precision interval packages: Comparing different approaches. In: Alt, R., Frommer, A., Kearfott, R.B., Luther, W. (eds.) Dagstuhl Seminar 2003. LNCS, vol. 2991, pp. 64–90. Springer, Heidelberg (2004)
10. Hearn, A.C.: REDUCE User's Manual, Version 3.8. RAND (2005)
11. Hözl, J.: Proving real-valued inequalities by computation in Isabelle/HOL. Diploma thesis, Institut für Informatik, Technische Universität München (April 2009)
12. Kohlhase, M., Lemburg, J., Schröder, L., Schulz, E.: Formal management of cad/cam processes. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 223–238. Springer, Heidelberg (2009)
13. Maplesoft. Maple 10 User Manual (2005)
14. Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007)
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
16. Schröder, L., Mossakowski, T.: HasCASL: Integrated higher-order specification and program development. *Theoret. Comput. Sci.* 410, 1217–1260 (2009)
17. Sofroniou, M., Spaletta, G.: Precise numerical computation. *J. Logic Algebraic Programming* 64(1), 113–134 (2005)
18. Technical Committee CEN/TC 74. EN 1591 – Flanges and their joints – Design rules for gasketed circular flange connections (2001)

Modelling Non-linear Crowd Dynamics in Bio-PEPA

Mieke Massink¹, Diego Latella¹, Andrea Bracciali^{1,3}, and Jane Hillston²

¹ Istituto di Scienza e Tecnologie dell'Informazione 'A. Faedo', CNR, Italy

² School of Informatics, University of Edinburgh, U.K.

³ Department of Computing Science and Mathematics, University of Stirling, U.K.

Abstract. Emergent phenomena occur due to the pattern of non-linear and distributed local interactions between the elements of a system over time. Surprisingly, agent based crowd models, in which the movement of each individual follows a limited set of simple rules, often re-produce quite closely the emergent behaviour of crowds that can be observed in reality. An example of such phenomena is the spontaneous self-organisation of drinking parties in the squares of cities in Spain, also known as "El Botellón" [20]. We revisit this case study providing an elegant stochastic process algebraic model in Bio-PEPA amenable to several forms of analyses, among which simulation and fluid flow analysis. We show that a fluid flow approximation, i.e. a deterministic reading of the average behaviour of the system, can provide an alternative and efficient way to study the same emergent behaviour as that explored in [20] where simulation was used instead. Besides empirical evidence, also an analytical justification is provided for the good correspondence found between simulation results and the fluid flow approximation.

Keywords: Fluid flow, process algebra, crowd dynamics, self-organisation.

1 Introduction

In modern society the formation of crowds, intended as large concentrations of people, is a phenomenon that occurs frequently. Well known examples are crowds at large entertainment events in cities or other open-air facilities such as sport stadiums, but also crowds at large airports and train stations. Fortunately, such crowds usually occur and dissolve without serious problems. However, in some cases accidents happen with possibly major consequences such as loss of lives and a large number of injuries [22]. The recent drama at the open-air festival in Germany is sadly adding to the list of such events [16].

There is an ever stronger interest in being able to prevent such disasters and there exists an extensive literature on numerous approaches to the study of crowd formation, crowd management and emergency egress [21]. Simulation models play an important role in these approaches. In particular, agent based modelling has become popular in recent years because it may provide valuable information about the dynamics of systems that contain non-linear elements,

chaos and random cause and effect. Several works in this area, e.g. work by Still [22], show that a crowd of people in which each individual follows a limited number of simple rules produces quite closely the emergent behaviour that can be observed in real human crowds. Emergent phenomena are known to occur due to the pattern of non-linear and distributed local interactions between the large number of elements of a system over time. His work and that of others have led to the development of several professional tools, based on agent simulation, for the realistic analysis and prediction of crowd behaviour in emergency situations. Such analyses may help to detect architectural or organisational problems that may potentially cause loss of lives in the event of emergency situations.

However, for very large crowds, analysis via detailed simulation may become costly and time-consuming since each execution of the model produces only a single trajectory through the state space whereas many executions are needed to reach statistically relevant conclusions. Such high costs may be justified when a final verification of a system is required, but often become prohibitive in situations in which a quick analysis is required to compare the consequences of different design options or when a large number of slightly different scenarios need to be analysed, i.e. when one may be interested in approximated, but efficient, analyses.

In a completely different field of research, namely that of the analysis of biochemical reactions, it has been shown that under certain conditions, such as the presence of a sufficiently large population, a deterministic continuous interpretation of models composed of many similar small independent components provide a good approximation of the average transient behaviour of the overall model. In the context of stochastic process algebras this insight has led to the development of an alternative formal fluid flow semantics for PEPA first and later for Bio-PEPA, a variant of PEPA originally devised for modelling biochemical processes. Such semantics are based on the generation of sets of ordinary differential equations (ODE) [12,4,7]. An application of PEPA with this alternative semantics in the context of emergency egress has been presented in [17,3]; it has been shown that the approach is an efficient and scalable alternative to simulation when average behaviour is of interest. It is not suitable to analyse, for example, exceptional or oscillatory behaviour. Good correspondence with results from the literature on evacuation times and node profiles—the average number of people present in a particular part of the building over time during egress—has been achieved. In that work only linear differential equations have been considered.

In this paper we revisit the case of self-organisation of crowds in a city as described by Rowe and Gomez in [20]. The work was inspired by a typical social phenomenon observed in Spanish cities, on summer nights, called “El Botellón”, when crowds of youngsters wander between city squares in search of a party. Such self-organising parties sometimes lead to heavy drinking and noisy behaviour until late at night. It turned out to be hard to predict when and where a large party would take place. The aim of the work by Rowe and Gomez was to gain insight into the general principles due to which parties self-organise, abstracting from specific details of individual cases. We show that with Bio-PEPA a fluid

flow approximation can provide an alternative way to study the same emergent behaviour as that explored in [20] where simulation was used instead. This case differs from that of emergency egress mentioned before because of the presence of non-linear aspects where the behaviour of the individual agents depends directly on other, similar agents present in the same environment. In [20] the movement of a crowd in a city is studied under various assumptions about the likelihood that people remain in a square. Agents follow two basic rules. The first rule defines when agents remain in a square, which depends on the “chat-probability”, i.e. the likelihood to meet someone in the square to chat with. The second rule defines how agents move between squares.

Rowe and Gomez, by developing an analytical model, determined a threshold of the chat-probability below which people are freely moving through the city and above which large crowds start to form. They validated their theory by the simulation of a multi-agent model for a ring topology of 4 squares and up to 80 agents. Both the theory and the simulation results show that for a value of the chat probability $c = n/N$, where n is the number of squares and N the number of agents, a clear phase-transition can be observed between a steady-state situation in which agents are evenly distributed over the squares (when c is below the threshold) and a situation in which agents spontaneously gather in one or a few squares (when c is above the threshold).

In this paper, we approach the modelling of crowds by adopting the Bio-PEPA stochastic process algebra [8]. Bio-PEPA embeds a notion of spatial location, intended to model compartments, suitable to describe the city topology and locate agents within it. Moreover, some aspects of the agent behaviour can be expressed as a function of the current state of the system, such as the number of people present in a square, an abstraction of the act of sensing the environment, common to standard agent models. Such a function may contain non-linear elements, which makes Bio-PEPA also particularly interesting for the analysis of some forms of emergent behaviour, as we will see in later sections. The fluid flow results obtained with the Bio-PEPA model correspond surprisingly well to the simulation results obtained with the the same model and to those published by Rowe and Gomez [20]. Informally speaking, for models where the rates can be expressed as functions of the average density of the population, under certain conditions, this phenomenon is well known (assuming that the populations are sufficiently large), see e.g. Kurtz [14] and in the context of mean field analysis Le Boudec et al. [2]. However, the rate functions in the crowds model addressed in this paper cannot be expressed this way. We provide an alternative analytical explanation for the observed correspondence which is partially based on recent work by Hayden and Bradley [11] on PEPA.

The outline of the paper is as follows. Section 2 recalls the crowd model used in the case study by Rowe and Gomez [20]. Section 3 briefly presents Bio-PEPA and its analysis environment. Section 4 describes the Bio-PEPA model of the collective behaviour of crowds in a city followed by a selection of the analyses results in Section 5. Section 6 provides insight in the close correspondence between the simulation results and fluid flow approximation for this case. Finally,

in Section 7 conclusions are presented and future research is outlined. A preliminary version of our results has been discussed at the PASTA 2010 workshop [19], whereas further results and details can be found in [18].

2 Rowe and Gomez Model of Crowd Dynamics

In this section we briefly recall the model of movement of crowds between squares in a city as presented by Rowe and Gomez in [20]. Assume a city with n squares represented as a graph with vertices $\{1, 2, \dots, n\}$. People are simulated by “agents” that are following a simple set of rules. The number of agents in square i , with $i \in \{0, 1, \dots, n\}$, at time t , with t representing discrete time steps, is represented by $p_i(t)$. The state of the system at t is given by the number of agents present in each square modelled by the vector $\mathbf{p}(t) = (p_1(t), p_2(t), \dots, p_n(t))$. The total number of agents N at any time t is constant: $N = \sum_{i=1}^n p_i(t)$.

Agents are located in squares. The rules guiding agents’ behaviour are the following. The probability that an agent decides to remain in a square depends on how many other agents are present in the same square. If square i contains $p_i > 0$ agents, the probability that an agent *leaves* the square is given by $(1 - c)^{p_i - 1}$. The parameter c (representing the *chat probability*, $0 \leq c \leq 1$) is the probability that an agent finds another one to talk to and thus remains in the square. Note that when there is only one agent in the square, it decides to leave with probability 1, since there is nobody else to talk to. If an agent decides to move, it moves with equal probability to any neighbouring square reachable by a street. Considering an analytical model of the above discrete behaviour the *expected* number of agents that will leave square i at a given time step t is given by the function:

$$f_i(t) = p_i(t)(1 - c)^{p_i(t) - 1}$$

This models the part of the population in square i that does not find anyone to talk to in that square¹. The probability that an agent, which decided to leave square j , moves to the adjacent square i is given by the matrix A_{ij} :

$$A_{ij} = \text{con}_{ij}/d_j$$

where d_j is the degree of vertex j , i.e. the number of streets departing from square j , and con_{ij} denotes that square i is connected to square j :

$$\text{con}_{ij} = \begin{cases} 1 & \text{if } i \text{ is connected to } j \\ 0 & \text{otherwise} \end{cases}$$

Clearly, $\text{con}_{ij} = \text{con}_{ji}$ and we assume that adjacent squares are connected by at most one street. The expected distribution of agents over squares at time $t + 1$ can now be defined as:

$$\mathbf{p}(t + 1) = \mathbf{p}(t) - \mathbf{f}(t) + A\mathbf{f}(t)$$

¹ Note that in this analytical model the number of agents $p_i(t)$ in square i is now approximated by a real number: the *expected* number of agents in square i at time t .

Clearly, from this formula it follows that a steady-state behaviour is reached when $\mathbf{f}(t) = A\mathbf{f}(t)$. In other words, when the number of people entering a square is equal to the number leaving the square. Rowe and Gomez show that there are two possibilities for such a stable state. In one case the agents freely move between squares and their distribution is proportional to the number of streets connected to each square. In the second case agents gather in large groups in a small number of squares corresponding to emergent self-organisation of parties. Which of the two situations will occur depends critically on the value of the chat probability c . When all squares have the same number of neighbouring squares a phase shift occurs at about $c = n/N$ where n is the number of squares and N the number of agents. For $c < n/N$ people freely move between squares whereas for $c > n/N$ agents self-organise into large groups. Simulation of the model confirms in an empirical way that this estimate for c is quite accurate when the population is large enough where large means about 60 agents or more in a 4-square topology.

For topologies where each square has the same number of streets the critical value of c can be estimated in an analytical way. For less regular topologies and when different squares have different chat probabilities and not all directions leaving from a square are equally likely to be taken by people it is very difficult to identify such critical values in an analytical way. Usually, in such cases simulation is used to analyse the models. However, when a large number of agents is involved, simulation may be extremely time consuming.

3 Bio-PEPA and Fluid Flow Analysis

In this section we briefly describe Bio-PEPA [7,8,6], a language that has recently been developed for the modelling and analysis of biochemical systems. The main components of a Bio-PEPA system are the “*species*” components, describing the behaviour of individual entities, and the *model component*, describing the interactions between the various species. The initial amounts of each type of entity or species are given in the model component.

The syntax of the Bio-PEPA components is defined as:

$$S ::= (\alpha, \kappa) \text{ op } S \mid S + S \mid C \quad \text{with op} = \downarrow \mid \uparrow \mid \oplus \mid \ominus \mid \odot \quad P ::= P \bowtie_c P \mid S(x)$$

where S is a *species component* and P is a *model component*. In the prefix term $(\alpha, \kappa) \text{ op } S$, κ is the *stoichiometry coefficient* of species S in action α . This arises from the original formulation of the process algebra for modelling biochemical reactions, where the stoichiometric coefficient captures how many molecules of a species are required for a reaction. However it may be interpreted more generally as the multiples of an entity involved in an occurring action. The default value of κ is 1 in which case we simply write α instead of (α, κ) . The *prefix combinator* “op” represents the role of S in the action, or conversely the impact that the action has on the species. Specifically, \downarrow indicates a *reactant* which will be consumed in the action, \uparrow a *product* which is produced as a result of the action, \oplus an *activator*, \ominus an *inhibitor* and \odot a generic *modifier*, all of which

play a role in an action without being produced or consumed and have a defined meaning in the biochemical context. The operator “+” expresses the choice between possible actions, and the constant C is defined by an equation $C=S$. The process $P \bowtie_{\mathcal{L}} Q$ denotes synchronisation between components P and Q , the set \mathcal{L} determines those actions on which the components P and Q are forced to synchronise, with \bowtie^* denoting a synchronisation on all common actions. In $S(x)$, the parameter $x \in \mathbb{R}$ represents the initial amount of the species.

A Bio-PEPA *system* with *locations* consists of a set of species components, also called sequential processes, a model component, and a context (locations, functional/kinetics rates, parameters, etc.). The prefix term $(\alpha, \kappa) \text{ op } S@l$ is used to specify that the action is performed by S in location l . The notation $\alpha[I \rightarrow J] \odot S$ is a shorthand for the pair of reactions $(\alpha, 1) \downarrow S@I$ and $(\alpha, 1) \uparrow S@J$ that synchronise on action α ². This shorthand is very convenient when modelling agents migrating from one location to another as we will see in the next section. Bio-PEPA is given an operational semantics [8] which is based on Continuous Time Markov Chains (CTMCs).

The Bio-PEPA language is supported by a suite of software tools which automatically process Bio-PEPA models and generate internal representations suitable for different types of analysis [8,5]. These tools include mappings from Bio-PEPA to differential equations (supporting a fluid flow approximation), stochastic simulation models [10], CTMCs with levels [7] and PRISM models [15].

A Bio-PEPA model describes a number of sequential components each of which represents a number of entities in a distinct state. The result of an action is to increase the number of some entities and decrease the number of others. Thus the total state of the system at any time can be represented as a vector with entries capturing the counts of each species component (i.e. an aggregated CTMC). This gives rise to a discrete state system which undergoes discrete events. The idea of fluid flow analysis is to approximate these discrete jumps by continuous flows between the states of the system.

4 Modelling Crowd Movement with Bio-PEPA

Let us consider the same small ring topology with 4 squares, A , B , C and D , as in Rowe and Gomez, allowing bi-directional movement between squares. The excerpt from the Bio-PEPA specification below defines this topology. The default compartment *top* contains all other compartments. The next line defines square A . Definitions for the other squares are similar and have been omitted. In this context *size* is used to denote a capacity in terms of number of agents.

```
location top : size = 1000, type = compartment;
location sqA in top : size = normal_square, type = compartment;
```

The size of the squares, defined by parameter *normal_square* = 100, is defined in such a way that all agents, 60 in this case, would fit in any single square and

² The concrete syntax for writing this in the Bio-PEPA tool set differs somewhat.

does not impose any further constraints. The Bio-PEPA specification, which we will henceforth refer to as the ‘crowd model’, has two further parameters. The parameter c defines the chat-probability and the parameter d the degree or number of streets connected to a square. In the considered topology $d = 2$ for each square. The actions modelling agents moving from square X to square Y will be denoted by $fXtY$. The associated functional rate (indicated by the keyword “kineticLawOf”) is defined in analogy to [20] (see Section 2). Since the only information on the probability distribution available is the expected number of agents leaving a square per time unit, this same information can also be modelled as the rate parameter of an exponential distribution. If one also considers the uniform distribution of people over the outgoing streets of the square then this rate needs to be divided by its degree d when agents leaving through a particular street are considered.

So the general rate with which agents leave square X via a particular street is:

$$(P@sqX * (1 - c)^{(P@sqX - 1)})/d$$

This leads to the following functional rates for the crowd model, one for each direction of movement. Only the one for $fAtB$ is shown, the others being similar:

$$\text{kineticLawOf } fAtB : (P@sqA * (1 - c)^{(P@sqA - 1)})/d;$$

The sequential component P below specifies the possible movements of a typical agent between squares. For example, $fAtB[sqA \rightarrow sqB] \odot P$ means that an agent present in square A moves to square B according to the functional rate defined for the action $fAtB$.

$$\begin{aligned} P = & fAtB[sqA \rightarrow sqB] \odot P + fBtA[sqB \rightarrow sqA] \odot P + \\ & fAtC[sqA \rightarrow sqC] \odot P + fCtA[sqC \rightarrow sqA] \odot P + \\ & fBtD[sqB \rightarrow sqD] \odot P + fDtB[sqD \rightarrow sqB] \odot P + \\ & fCtD[sqC \rightarrow sqD] \odot P + fDtC[sqD \rightarrow sqC] \odot P; \end{aligned}$$

Finally, the model component defines the initial conditions of an experiment, i.e. in which squares the agents are located initially, and the relative synchronisation pattern. Initially, there are 60 agents in square A . This is expressed by $P@sqA[60]$ in the composition shown below. All other squares are initially empty (i.e. $P@sqX[0]$ for $X \in \{B, C, D\}$). The fact that moving agents need to synchronise follows from the definition of the shorthand operator \rightarrow .

$$(P@sqA[60] \underset{*}{\bowtie} P@sqB[0]) \underset{*}{\bowtie} (P@sqC[0] \underset{*}{\bowtie} P@sqD[0])$$

The total number of agents $P@sqA + P@sqB + P@sqC + P@sqD$ is invariant and amounts to 60 in this specific case.

5 Selected Results for a Model with Four Squares

This section presents a selection³ of the analysis results for the model with four squares. The figures report both analysis via Gillespie stochastic simulation

³ Further results can be found in [18].

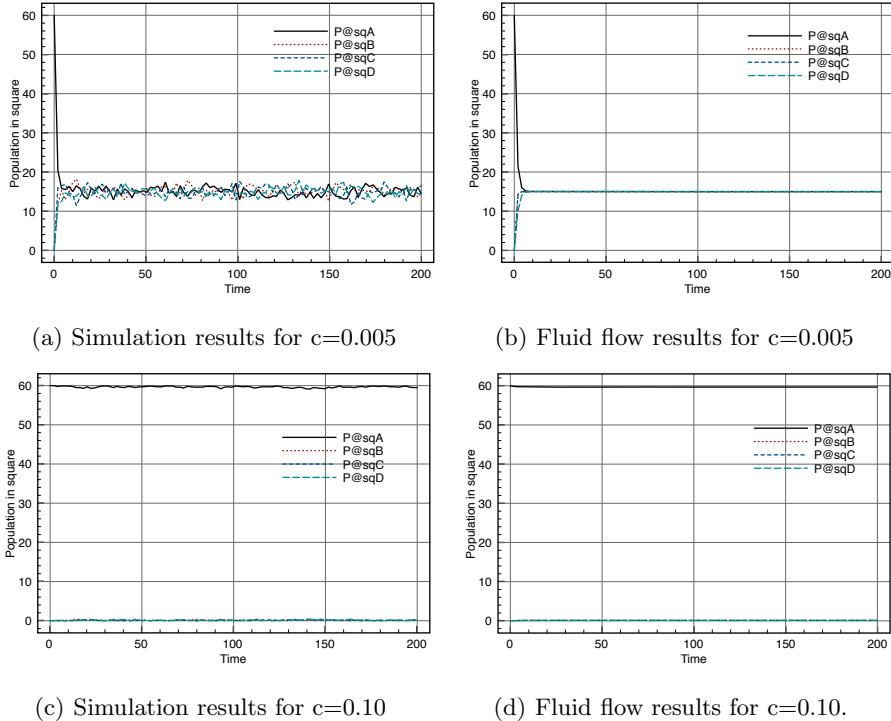


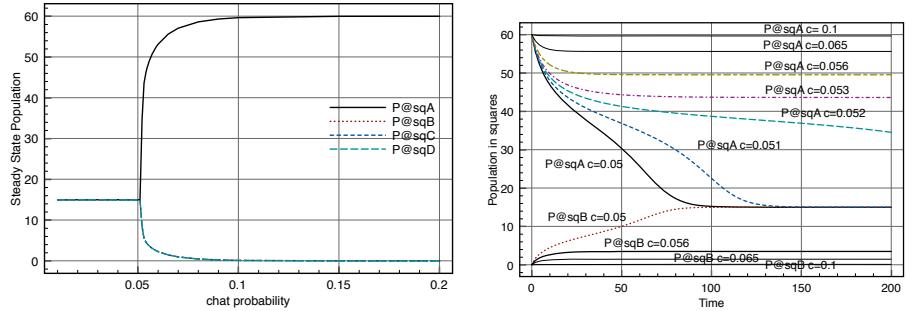
Fig. 1. Results for four squares with 60 agents in A initially

(G) [10], averaged over 10 independent runs, and fluid flow analysis based on the adaptive numeric solution of sets of ODEs based on the adaptive step-size 5th order Dormand-Prince ODE solver [9].⁴

Fig. 1(a) and Fig. 1(b) show stochastic simulation and fluid flow results for a model with 60 agents initially in square A and for $c = 0.005$, which is below the analytically estimated threshold of $c = n/N = 4/60 = 0.06666$. The results show that a dynamic equilibrium is reached, i.e. all agents distribute evenly over the four squares. This confirms the discrete event simulation results reported by Rowe and Gomez. Fig. 1(c) shows the results of stochastic simulation for the same model, but for $c = 0.10$, a value above the threshold. Corresponding fluid flow results for the same value of c are shown in Fig. 1(d). The figures show that the population settles rather quickly in a steady state in which almost all agents remain in square A. This is the second type of steady state observed also by Rowe and Gomez. Interestingly, the fluid flow analyses of the same model, for both values of c (Fig. 1(b) and Fig. 1(d)) show very good correspondence to the respective simulation results (Fig. 1(a) and Fig. 1(c), resp.).

Since these results show that both types of steady state emerge in this stochastic version of the crowd model and for both types of analysis, the question

⁴ All analyses have been performed with the Bio-PEPA Eclipse Plug-in tool [5] on a Macintosh PowerPC G5.



(a) Fluid flow results about “Steady state” population levels in each square (b) Fluid flow results for square A (and partially B) for varying chat probabilities

Fig. 2. Results at $t=200$ for varying chat probabilities

naturally arises whether fluid flow could be used as an efficient technique to investigate the behaviour of the model for various values of the chat-probability c , in particular those close to the critical threshold. Fig. 2(a) shows the expected number of agents in the squares at $t = 200$, starting with 60 agents in square A initially, for different chat probabilities ranging from 0.01 to 0.2 with steps of 0.01 (except between 0.05 and 0.065 where the steps are 0.001) in a *fluid flow analysis*. The figure shows clearly that for a chat probability below 0.05 in the steady state the population is evenly distributed over the four squares. For $c > 0.05$ the situation changes sharply. For these higher values of c the agent population tends to concentrate in square A, the square from which they all started. All other squares remain essentially empty. The results in Fig. 2(a) illustrate a clear case of spontaneous *self-organisation* or *emergent* behaviour. The results in Fig. 2(a) closely correspond to those obtained by Rowe and Gomez [20] by discrete event simulation. The ease and computational efficiency with which these results can be produced by means of fluid flow analysis opens up a promising perspective on how process algebraic fluid flow analysis could be used as an alternative, efficient, scalable and formal approach to investigate emergent behaviour in the vicinity of critical parameter values for this class of models.

An impression of how the distribution of agents over the four squares evolves for values of c that are close to the threshold of $c = 0.05$ is shown in Fig. 2(b). For $c = 0.05$ and $c = 0.051$ the agents still distribute uniformly over the four squares, though this takes a bit more time than for lower values of c . For $c = 0.052$ this situation is changing, and for $c = 0.053$ and higher clearly a different steady state is reached in which most agents group in the single square A. Note that for $c = 0.052$ a stable state has not yet been reached at time $t = 200$. However, this does not influence the overall picture. The theory on the stability of fixed-points predicts that the steady state behaviour for values of $c > n/N = 4/60 = 0.066666$ is unstable (see [20]). Instability in this case means extreme sensitivity to the initial values of the number of agents in each square. This phenomenon is illustrated by the results in Fig. 3 where single simulation runs are shown for the same model and the same initial conditions, i.e. 30 agents per square and

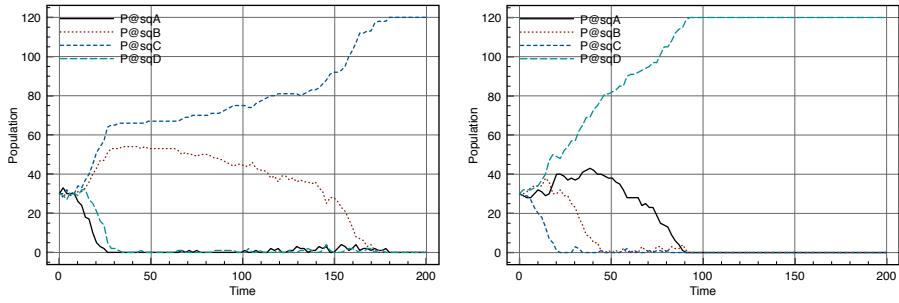
(a) Simulation results for $c=0.1$ (b) Simulation results for $c=0.1$

Fig. 3. Two single simulation runs for the same model with initially 30 agents in each square

$c = 0.1$. In every run the agents may all gather in one of the squares without leaving again. However, no prediction is possible on which square this will be except that they are all equally likely in this particular topology. Scalability of the fluid flow analysis is well illustrated by comparing the time of a *single* simulation run for 600,000 agents over 500 time units in a 3 by 3 grid topology, taking approx. 27 min., and a fluid flow analysis of that model taking 20 ms.⁵

6 Analytical Assessment of the Fluid Flow Approximation

The results in the previous sections show a very good correspondence between the results obtained via fluid flow approximation and those obtained, on the one hand, with Gillespie's stochastic simulation algorithm applied on exactly the same Bio-PEPA specification and, on the other hand, with discrete event simulation results found by Rowe and Gomez [20]. In this section we provide a justification for this correspondence from an analytical perspective.

There exist several theories that address the relation between the interpretation of the model as a large set of individual, independently behaving and interacting agents, as is the case for simulation, and a continuous deterministic interpretation of the same model as occurs with a fluid flow approximation. Perhaps the most well-known is the theory by Kurtz [14]. Informally speaking, Kurtz shows an exact relation (in the limit when the population goes to infinity) between the two above mentioned interpretations when the rate-functions can be expressed in terms of the average *density* of the population. A similar requirement needs to be satisfied in the context of the theory of mean field analysis, for example in recent work by Le Boudec et al. [2]. Unfortunately, in the crowd model the rate-functions cannot be expressed in terms of the density of the population because the exponent of the factor $(1 - c)^{(px - 1)}$ requires the absolute number px of agents present in square X .

⁵ These timing results have been obtained with the Bio-PEPA plugin tool for Eclipse Helios on a MacPro4,1.

A third approach, by Hayden and Bradley [11], has recently been applied to assess the quality of the fluid approximation for (grouped) PEPA models. In that approach the Chapman-Kolmogorov forward equations (C-K) are derived from a typical central state of the aggregated CTMC⁶ associated with a PEPA model. These equations are then used in the moment generating function from which, by partial differentiation, ordinary differential equations are obtained for the expected value over time of each sequential component in the PEPA model. In this section we adapt the approach to the Bio-PEPA crowds model which is characterised by non-linear rate functions. Let p_A , p_B , p_C and p_D

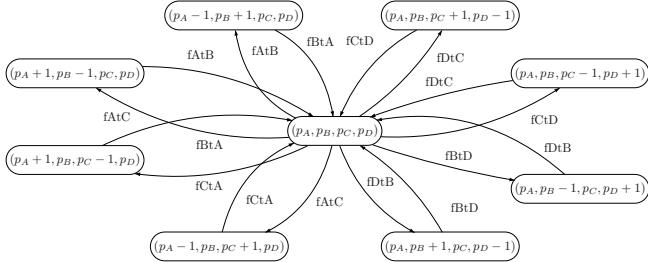


Fig. 4. A central state of the crowds model

denote the number of agents in square A , B , C and D , respectively. A central state of the aggregated CTMC of the crowds model is shown in Fig. 4. Let $\text{prob}_{(p_A, p_B, p_C, p_D)}(t)$ denote the transient probability of being in the aggregated CTMC state with p_X agents in square X , for X in $\{A, B, C, D\}$ at time t . From the crowd model in Sect. 4 we obtain the following C-K, governing the evolution of the state probabilities over time of the underlying aggregated CTMC:

$$\begin{aligned} \frac{d \text{prob}_{(p_A, p_B, p_C, p_D)}(t)}{dt} = & ((p_A + 1) \cdot (1 - c)^{p_A})/2 \cdot \text{prob}_{(p_A + 1, p_B - 1, p_C, p_D)}(t) \\ & + ((p_B + 1) \cdot (1 - c)^{p_B})/2 \cdot \text{prob}_{(p_A - 1, p_B + 1, p_C, p_D)}(t) \\ & + ((p_C + 1) \cdot (1 - c)^{p_C})/2 \cdot \text{prob}_{(p_A, p_B, p_C + 1, p_D - 1)}(t) \\ & + ((p_D + 1) \cdot (1 - c)^{p_D})/2 \cdot \text{prob}_{(p_A, p_B, p_C - 1, p_D + 1)}(t) \\ & + ((p_A + 1) \cdot (1 - c)^{p_A})/2 \cdot \text{prob}_{(p_A + 1, p_B, p_C - 1, p_D)}(t) \\ & + ((p_C + 1) \cdot (1 - c)^{p_C})/2 \cdot \text{prob}_{(p_A - 1, p_B, p_C + 1, p_D)}(t) \\ & + ((p_B + 1) \cdot (1 - c)^{p_B})/2 \cdot \text{prob}_{(p_A, p_B + 1, p_C, p_D - 1)}(t) \\ & + ((p_D + 1) \cdot (1 - c)^{p_D})/2 \cdot \text{prob}_{(p_A, p_B - 1, p_C, p_D + 1)}(t) \\ & - ((p_A) \cdot (1 - c)^{p_A - 1}) \cdot \text{prob}_{(p_A, p_B, p_C, p_D)}(t) \\ & - ((p_B) \cdot (1 - c)^{p_B - 1}) \cdot \text{prob}_{(p_A, p_B, p_C, p_D)}(t) \\ & - ((p_C) \cdot (1 - c)^{p_C - 1}) \cdot \text{prob}_{(p_A, p_B, p_C, p_D)}(t) \\ & - ((p_D) \cdot (1 - c)^{p_D - 1}) \cdot \text{prob}_{(p_A, p_B, p_C, p_D)}(t) \end{aligned}$$

Each of the eight first summands appears only when the state (p_A, p_B, p_C, p_D) has the corresponding incoming transitions in the aggregated state space.

⁶ For a formal definition see for example [11].

Let us now consider, following the approach outlined in [11], how an ODE for the function $P_A(t)$ can be obtained. First note that the expected value of $P_A(t)$ is given by:

$$\mathbb{E}[P_A(t)] = \sum_{(p_A, p_B, p_C, p_D)} p_A \cdot prob_{(p_A, p_B, p_C, p_D)}(t)$$

So this leads to the following derivation:

$$\begin{aligned} \frac{d\mathbb{E}[P_A(t)]}{dt} &= \{\text{By def. of expected value}\} \\ \frac{d \sum_{(p_A, p_B, p_C, p_D)} p_A \cdot prob_{(p_A, p_B, p_C, p_D)}(t)}{dt} &= \{\text{By distribution of differentiation}\} \\ \sum_{(p_A, p_B, p_C, p_D)} p_A \cdot \frac{d prob_{(p_A, p_B, p_C, p_D)}(t)}{dt} &= \{\text{By definition of C-K equations}\} \\ \sum_{(p_A, p_B, p_C, p_D)} [& ((p_A - 1) \cdot p_A \cdot (1 - c)^{(p_A - 1)}) / 2 \cdot prob_{(p_A, p_B, p_C, p_D)}(t) \\ & + ((p_A + 1) \cdot p_B \cdot (1 - c)^{(p_B)}) / 2 \cdot prob_{(p_A, p_B, p_C, p_D)}(t) \\ & + (p_A \cdot p_C \cdot (1 - c)^{(p_C - 1)}) / 2 \cdot prob_{(p_A, p_B, p_C, p_D)}(t) \\ & + (p_A \cdot p_D \cdot (1 - c)^{(p_D - 1)}) / 2 \cdot prob_{(p_A, p_B, p_C, p_D)}(t) \\ & + ((p_A - 1) \cdot p_A \cdot (1 - c)^{(p_A - 1)}) / 2 \cdot prob_{(p_A, p_B, p_C, p_D)}(t) \\ & + ((p_A + 1) \cdot p_C \cdot (1 - c)^{(p_C - 1)}) / 2 \cdot prob_{(p_A, p_B, p_C, p_D)}(t) \\ & + (p_A \cdot p_B \cdot (1 - c)^{(p_B - 1)}) / 2 \cdot prob_{(p_A, p_B, p_C, p_D)}(t) \\ & + (p_A \cdot p_D \cdot (1 - c)^{(p_D - 1)}) / 2 \cdot prob_{(p_A, p_B, p_C, p_D)}(t) \\ & - (p_A \cdot p_A \cdot (1 - c)^{(p_A - 1)}) \cdot prob_{(p_A, p_B, p_C, p_D)}(t) \\ & - (p_A \cdot p_B \cdot (1 - c)^{(p_B - 1)}) \cdot prob_{(p_A, p_B, p_C, p_D)}(t) \\ & - (p_A \cdot p_C \cdot (1 - c)^{(p_C - 1)}) \cdot prob_{(p_A, p_B, p_C, p_D)}(t) \\ & - (p_A \cdot p_D \cdot (1 - c)^{(p_D - 1)}) \cdot prob_{(p_A, p_B, p_C, p_D)}(t)] \end{aligned}$$

If $(P_A(t), P_B(t), P_C(t), P_D(t))$ is the state of the aggregated CTMC at time t , then by cancelling terms in the above equation one obtains:

$$\begin{aligned} \sum_{(p_A, p_B, p_C, p_D)} [& -p_A \cdot (1 - c)^{(p_A - 1)} \cdot prob_{(p_A, p_B, p_C, p_D)}(t) \\ & + (p_B \cdot (1 - c)^{(p_B - 1)} \cdot prob_{(p_A, p_B, p_C, p_D)}(t)) / 2 \\ & + (p_C \cdot (1 - c)^{(p_C - 1)} \cdot prob_{(p_A, p_B, p_C, p_D)}(t)) / 2 \end{aligned}$$

This yields in terms of expectations the following ODE for $\mathbb{E}[P_A(t)]$:

$$\begin{aligned} \frac{d\mathbb{E}[P_A(t)]}{dt} &= -\mathbb{E}[P_A(t) \cdot (1 - c)^{(\mathbb{E}[P_A(t)] - 1)}] \\ &+ (\mathbb{E}[P_B(t) \cdot (1 - c)^{(\mathbb{E}[P_B(t)] - 1)}]) / 2 \\ &+ (\mathbb{E}[P_C(t) \cdot (1 - c)^{(\mathbb{E}[P_C(t)] - 1)}]) / 2 \end{aligned}$$

If at this point, as shown in [11], the functions $(1 - c)^{(X-1)}$ were just constant rates, then expectation would just distribute over multiplication and one would obtain an equation in terms of expectations of populations. However, in our case the rate is a more complicated function and, in general, expectation does not distribute over an arbitrary function, i.e. $\mathbb{E}[\phi(X)] \neq \phi(\mathbb{E}[X])$. This means that exact equality cannot be obtained this way. As an alternative we consider whether $\mathbb{E}[P_A(t)](1 - c)^{(\mathbb{E}[P_A(t)] - 1)}$ could be expected to approximate $\mathbb{E}[P_A(t)]$.

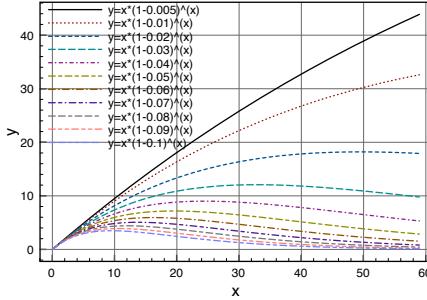


Fig. 5. Function $x(1 - c)^x$ for up to 60 agents and different chat probabilities

$(1 - c)^{(\mathbb{E}[P_A(t)] - 1)}$. To address this question we recall Jensen's work [13] from which it is known that for strictly convex functions ϕ the following inequality holds:

$$\mathbb{E}[\phi(X)] \geq \phi(\mathbb{E}[X])$$

with *equality* when ϕ is linear or when it is a constant. The reverse inequality holds in case ϕ is strictly concave. So this requires a closer investigation of the particular function at hand. In Fig. 5 graphs are shown of the function $\phi(x) = x(1 - c)^x$ for values of $x \in [0 \dots 60]$ and for various values of chat probability c . It can be observed that for very small values of c the function is almost linear for the number of agents considered. For larger values of c the function is mostly concave and tends to an almost constant value. This implies that, informally speaking, any hypothetical probability distribution of the values of x would be mapped on an increasingly “shrinking” version of the distribution of $\phi(x)$. This means that $\mathbb{E}[P_A(t)](1 - c)^{(\mathbb{E}[P_A(t)] - 1)}$ approximates $\mathbb{E}[P_A(t) \cdot (1 - c)^{(\mathbb{E}[P_A(t)] - 1)}]$ indeed rather well.

Distributing expectation over the function the following ODE for the expected value of $P_A(t)$ is obtained:

$$\begin{aligned} \frac{d\mathbb{E}[P_A(t)]}{dt} &\approx -\mathbb{E}[P_A(t)](1 - c)^{(\mathbb{E}[P_A(t)] - 1)} \\ &+ (\mathbb{E}[P_B(t)](1 - c)^{(\mathbb{E}[P_B(t)] - 1)})/2 \\ &+ (\mathbb{E}[P_C(t)](1 - c)^{(\mathbb{E}[P_C(t)] - 1)})/2 \end{aligned}$$

This ODE is identical to the one generated by the Bio-PEPA toolset which follows the approach described by Ciocchetta and Hillston in [7]. The derivation is provided in [18]. In a similar way the ODEs for the other stochastic variables can be obtained. This explains why the results obtained in this paper for fluid flow and for stochastic simulation also in this non-linear setting correspond so closely.

7 Conclusions and Further Work

The modelling and analysis of crowd dynamics appears to be an active and open research topic. We have explored the application of the stochastic process algebra

Bio-PEPA to model a simple but interesting non-linear case study concerning the emergent self-organisation of parties in the squares of a city. Bio-PEPA is based on a modular, high-level language providing notions of locality and context dependency. These features make Bio-PEPA also a promising candidate for the modelling of a class of possibly non-linear systems that goes beyond the bio-molecular applications it was originally designed for [3,8,1]. In this case study fluid flow approximation provides a computationally efficient analysis of the number of people, on average, that are present in the various squares when time evolves. The results are shown to correspond well to those found in the literature where they were obtained by means of more elaborate and time-consuming discrete event simulation. Also, an analytical approach to explain this good correspondence for a model the rates of which cannot be expressed as functions of the average population density, has been provided. Although the simple topology addressed in this paper, chosen for reasons of validation of the approach, can be analysed analytically, more complex topologies and more realistic models, addressing issues such as the influence of the presence of friends on people's behaviour and the attractiveness of squares, may easily turn out to be too complex to be studied analytically. Fluid flow approximation with Bio-PEPA has proved to be a suitable choice in this case. Preliminary results about more complex models can be found in [18].

Future work is developing along a few main directions. We are interested in developing further linguistic abstractions to describe more precisely the dynamics of systems with a large number of mobile agents displaced in a, possibly open, physical environment. We are furthermore interested in conducting more fundamental research on the fluid flow approach and its relationship to emergent non-linear behaviour, in particular its relation to mean field analysis [2].

Acknowledgments. The authors would like to thank Stephen Gilmore, Maria Luisa Guerriero, Allan Clark and Adam Duguid (University of Edinburgh) for their support with the Bio-PEPA plug-in, Richard Hayden and Jeremy Bradley (Imperial College London) for discussions on approximations and Michael Harrison and Nigel Thomas for the case study. This research has been partially funded by the CNR project RSTL-XXL and by the EU-IP project ASCENS (nr. 257414). Jane Hillston has been supported by the EPSRC ARF EP/c543696/01.

References

1. Akman, O.E., Ciocchetta, F., Degasperi, A., Guerriero, M.L.: Modelling biological clocks with bio-PEPA: Stochasticity and robustness for the *neurospora crassa* circadian network. In: Degano, P., Gorrieri, R. (eds.) CMSB 2009. LNCS, vol. 5688, pp. 52–67. Springer, Heidelberg (2009)
2. Benaïm, M., Le Boudec, J.: A class of mean field interaction models for computer and communication systems. Performance Evaluation 65(11-12), 823–838 (2008)
3. Bracciali, A., Hillston, J., Latella, D., Massink, M.: Reconciling population and agent models for crowd dynamics. In: Proceedings of 3rd International Workshop on Logics, Agents, and Mobility, LAM 2010 (to appear, 2010)

4. Calder, M., Gilmore, S., Hillston, J.: Automatically deriving odes from process algebra models of signalling pathways. In: Plotkin, G. (ed.) Proceedings of Computational Methods in Systems Biology (CMSB 2005), pp. 204–215 (2005)
5. Ciocchetta, F., Duguid, A., Gilmore, S., Guerriero, M.L., Hillston, J.: The Bio-PEPA Tool Suite. In: Proc. of the 6th Int. Conf. on Quantitative Evaluation of SysTems (QEST 2009), pp. 309–310 (2009)
6. Ciocchetta, F., Guerriero, M.L.: Modelling biological compartments in Bio-PEPA. ENTCS 227, 77–95 (2009)
7. Ciocchetta, F., Hillston, J.: Bio-PEPA: An extension of the process algebra pepa for biochemical networks. ENTCS 194(3), 103–117 (2008)
8. Ciocchetta, F., Hillston, J.: Bio-PEPA: A framework for the modelling and analysis of biological systems. TCS 410(33-34), 3065–3084 (2009)
9. Dormand, J.R., Prince, P.J.: A family of embedded Runge-Kutta formulae. Journal of Computational and Applied Mathematics 6(1), 19–26 (1980)
10. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. The Journal of Physical Chemistry 81(25), 2340–2361 (1977)
11. Hayden, R.A., Bradley, J.T.: A fluid analysis framework for a Markovian process algebra. TCS 411(22-24), 2260–2297 (2010)
12. Hillston, J.: Fluid flow approximation of PEPA models. In: Proceedings of QEST 2005, pp. 33–43. IEEE Computer Society Press, Los Alamitos (2005)
13. Jensen, J.L.W.V.: Sur les fonctions convexes et les inégalités entre les valeurs moyennes. Acta Mathematica 30(1), 175–193 (1906)
14. Kurtz, T.G.: Solutions of ordinary differential equations as limits of pure Markov processes. Journal of Applied Probability 7(1), 49–58 (1970)
15. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic model checking for performance and reliability analysis. ACM SIGMETRICS Performance Evaluation Review (2009)
16. Love-parade: Stampede at german love parade kills 19,
<http://www.bbc.co.uk/news/world-europe-10751899>, (accessed on August 10, 2010)
17. Massink, M., Latella, D., Bracciali, A., Harrison, M.: A scalable fluid flow process algebraic approach to emergency egress analysis. In: Proceedings of the 8th International Conference on Software Engineering and Formal Methods (SEFM 2010), pp. 169–180. IEEE, Los Alamitos (2010)
18. Massink, M., Latella, D., Bracciali, A., Hillston, J.: A combined process algebraic, agent and fluid flow approach to emergent crowd behaviour. Tech. Rep. 2010-TR-025, CNR-ISTI (2010)
19. Massink, M., Latella, D., Bracciali, A., Hillston, J.: Modelling crowd dynamics in Bio-PEPA – extended abstract. In: Participant Proceedings of the 9th Workshop on Process Algebra and Stochastically Timed Activities, PASTA 2010 (2010)
20. Rowe, J.E., Gomez, R.: El Botellón: Modeling the movement of crowds in a city. Complex Systems 14, 363–370 (2003)
21. Santos, G., Aguirre, B.E.: A critical review of emergency evacuation simulation models. In: Proceedings of the NIST Workshop on Building Occupant Movement during Fire Emergencies, June 10-11, 2004, pp. 27–52. NIST/BFRL Publications Online, Gaithersburg, MD, USA (2005)
22. Still, G.K.: Crowd dynamics (2000), Ph. D. Thesis, University of Warwick, U.K..

Smart Reduction

Pepijn Crouzen¹ and Frédéric Lang²

¹ Computer Science, Saarland University, Saarbrücken, Germany
`crouzen@cs.uni-saarland.de`

² VASY project team, INRIA Grenoble Rhône-Alpes/LIG, Montbonnot, France
`Frederic.Lang@inria.fr`

Abstract. Compositional aggregation is a technique to palliate state explosion — the phenomenon that the behaviour graph of a parallel composition of asynchronous processes grows exponentially with the number of processes — which is the main drawback of explicit-state verification. It consists in building the behaviour graph by incrementally composing and minimizing parts of the composition modulo an equivalence relation. Heuristics have been proposed for finding an appropriate composition order that keeps the size of the largest intermediate graph small enough. Yet the underlying composition models are not general enough for systems involving elaborate forms of synchronization, such as multiway and/or nondeterministic synchronizations. We overcome this by proposing a generalization of compositional aggregation that applies to an expressive composition model based on synchronization vectors, subsuming many composition operators. Unlike some algebraic composition models, this model enables any composition order to be used. We also present an implementation of this approach within the CADP verification toolbox in the form of a new operator called *smart reduction*, as well as experimental results assessing the efficiency of smart reduction.

1 Introduction

Explicit-state verification is a way of ascertaining whether a system fulfills its specification, by systematically exploring its behaviour graph. The main limitation of explicit-state verification is the exponential growth of the behaviour graph, known as state explosion. For systems consisting of asynchronous processes executing in parallel, *compositional aggregation* [11] (also known as *incremental reachability analysis* [29], *compositional state space minimization* [32,20,25], and *compositional reachability analysis* [9,19]) is a way to palliate state explosion by incrementally *aggregating* (i.e., composing and then minimizing modulo an equivalence relation) parts of the system. Compositional aggregation was applied successfully to systems from various domains [8,22,15,31,5,4,6].

Due to their modular nature, software systems are appropriate for compositional modeling and verification. Examples of studies include software reuse [12], unit testing [30], web service performance [13], middleware specification [28], software deployment protocols [31], multi-processor multi-threaded architectures [10], and software decomposition [7], in which processes usually represent

software components, such as servers, packages, threads, objects or functions. These applications often involve the (possibly automatic) translation of (architectural) software description languages such as UML, statecharts, or BPEL, each of which provides its own composition model, to a formal model.

The efficiency of compositional aggregation depends on the order in which the concurrent processes are aggregated. In practice, the order is often specified by the designer of a concurrent system, either explicitly, or implicitly through the order and hierarchy of the concurrent processes. Since it is not possible to know precisely whether an order will be more or less efficient than another without trying them and comparing the results, the user generally has to rely on intuition. This task is difficult for large and/or not hierarchical compositions, and impractical for compositional models that are automatically generated from a higher-level description.

Heuristics to automatically determine efficient aggregation orders, based on the process interactions, have been proposed in [29] for concurrent finite state machines communicating via named channels. More recently, such heuristics have been refined and implemented in a prototype tool for processes synchronizing on their common alphabets [11]. In both works, the processes to be composed are selected using two metrics: an estimate of the proportion of internal transitions in the composition (the higher, the more the composition graph being expected to be reducible), and an estimate of the proportion of transitions that interleave. A limitation of the above works lies in the limited forms of synchronizations enabled by their composition models, which are generally insufficient to capture the semantics of the composition models of state-of-the-art software description languages: The composition model used in [29] does not enable multiway synchronization (more than two processes synchronizing all together), and neither of the composition models used in [29,11] enables nondeterministic synchronization (a process synchronizing with one or another on a given label).

This paper presents a refinement of the compositional aggregation techniques of [29,11], called *smart reduction*. Smart reduction uses an expressive composition model named *networks of LTss* (*Labeled Transition Systems*) [26], inspired by synchronization vectors in the style of MEC [1] and FC2 [3], which has two major advantages. The first advantage is that it makes the compositional aggregation technique more general: networks of LTss subsume not only the models used in [29,11], but also many other concurrent operators. They include the parallel composition, label hiding, label renaming and label cutting (sometimes also called label restriction) found in process algebras (e.g., CCS [27], CSP [23], LOTOS [24], μ CRL [21], etc.). They also include more general parallel composition such as that of E-LOTOS/LOTOS NT [18], which enables n among m synchronization (any n processes synchronizing together among a set of m) and synchronization by interfaces (all processes sharing a label in their interface synchronizing together on that label). The latter operators have been shown to be expressive enough to reflect the graphical structure of process networks [18], such as those found in graphical software description languages. In particular, synchronization by interfaces was adopted in the FIACRE intermediate model for

avionic systems [2]. The second advantage is that networks enable any aggregation order, which is not in general the case in process algebraic models, where some composition orders, possibly including the optimal order, may not be representable using the available algebraic operators. This paper also presents the implementation of smart reduction in the CADP toolbox [17], and experimental results that assess the effectiveness of smart reduction on several case studies.

Paper overview. Networks of LTSS are defined in Section 2. Compositional aggregation of networks is described and illustrated in Section 3. Metrics for selecting a good aggregation order are presented in Section 4. The implementation within CADP is described in Section 5. Experimentation on existing case studies is reported in Section 6. Finally, concluding remarks are given in Section 7.

2 Networks of LTSS

The *network of LTSS* model (or *networks* for short) was introduced in [26] as an intermediate model to represent compositions of LTSS using various operators. We first give a few background definitions before defining the model formally.

Background. Given two integers n and m , we write $n..m$ for the set of integers ranging from n to m . If $n > m$ then $n..m$ denotes the empty set. A *vector* \mathbf{v} of size n is a set of n elements indexed by $1..n$. For $i \in 1..n$, we write $\mathbf{v}[i]$ for the element of \mathbf{v} at index i . We write $()$ for the vector of size 0, (e_1) for the vector \mathbf{v} of size 1 such that $\mathbf{v}[1] = e_1$, and more generally (e_1, \dots, e_n) for the vector \mathbf{v} of size n such that $(\forall i \in 1..n) \mathbf{v}[i] = e_i$. Given \mathbf{v}_1 , a vector of size n_1 , and \mathbf{v}_2 , a vector of size n_2 , $\mathbf{v}_1 \oplus \mathbf{v}_2$ denotes the vector of size $n_1 + n_2$ obtained by concatenation of \mathbf{v}_1 and \mathbf{v}_2 , defined by $(\forall i \in 1..n_1) (\mathbf{v}_1 \oplus \mathbf{v}_2)[i] = \mathbf{v}_1[i]$ and $(\forall i \in n_1 + 1..n_1 + n_2) (\mathbf{v}_1 \oplus \mathbf{v}_2)[i] = \mathbf{v}_2[i - n_1]$. The expression $e :: \mathbf{v}$ denotes adjunction of e to the head of \mathbf{v} and is defined as $(e) \oplus \mathbf{v}$. Given an ordered subset of $1..n$ I , such that $I = \{i_1, \dots, i_m\}$ with $i_1 < \dots < i_m$ ($0 \leq m \leq n$), $\mathbf{v}|_I$ denotes the projection of \mathbf{v} on to the set of indexes I , defined as the vector of size m such that $(\forall j \in 1..m) \mathbf{v}|_I[j] = \mathbf{v}[i_j]$. We write as \bar{I} the set $1..n \setminus I$. For any set S , we write $|S|$ for the number of elements of S . An LTS (*Labeled Transition System*) is a tuple $(\Sigma, A, \longrightarrow, s_0)$, where Σ is a set of states, A is a set of labels, $\longrightarrow \subseteq \Sigma \times A \times \Sigma$ is the (labeled) transition relation, and $s_0 \in \Sigma$ is the initial state.

Networks of LTSS. A *network of LTSS* N of size n is a pair (\mathbf{S}, V) where:

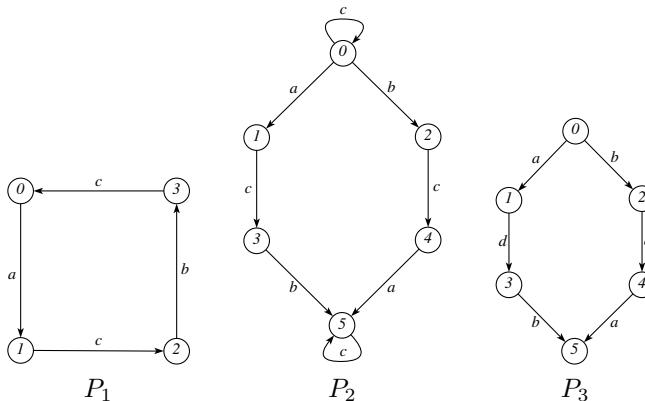
- \mathbf{S} is a vector of LTSS (called *individual LTSS*) of size n . We write respectively \longrightarrow_i , Σ_i , and s_i^0 for the transition relation, the set of states, and the initial state of $\mathbf{S}[i]$. For a label b , we also write \xrightarrow{b}_i for the largest subset of \longrightarrow_i containing only transitions labeled by b .
- V is a finite set of *synchronization rules*. Each synchronization rule has the form (\mathbf{t}, a) , where a is a label and \mathbf{t} is a vector of size n , called a *synchronization vector*, whose elements are labels and occurrences of a special symbol \bullet that does not occur as a label in any individual LTS.

To a network N can be associated a (global) LTS $lts(N)$ which is the parallel composition of its individual LTSSs. Each rule $(\mathbf{t}, a) \in V$ defines transitions labeled by a , obtained either by synchronization (if several indices i are such that $\mathbf{t}[i] \neq \bullet$) or by interleaving (otherwise) of individual LTS transitions. Formally, $lts(N)$ is defined as the LTS $(\Sigma, A, \longrightarrow, \mathbf{s}_0)$, where $\Sigma = \Sigma_1 \times \dots \times \Sigma_n$, $A = \{a \mid (\mathbf{t}, a) \in V\}$, $\mathbf{s}_0 = (s_1^0, \dots, s_n^0)$, and \longrightarrow is the smallest transition relation satisfying:

$$(\mathbf{t}, a) \in V \wedge (\forall i \in 1..n) (\mathbf{t}[i] = \bullet \wedge s'[i] = s[i]) \vee (\mathbf{t}[i] \neq \bullet \wedge s[i] \xrightarrow{\mathbf{t}[i]} s'[i]) \Rightarrow s \xrightarrow{a} s'$$

If $\mathbf{t}[i] \neq \bullet$, we say that $S[i]$ is *active* for the rule (\mathbf{t}, a) , otherwise we say that $S[i]$ is *inactive*. We write $A(\mathbf{t})$ for the set of individual LTS indexes active for a rule, defined as $\{i \mid i \in 1..n \wedge \mathbf{t}[i] \neq \bullet\}$. We say that a rule (\mathbf{t}, a) or a synchronization vector \mathbf{t} is *controlled* by LTS $S[i]$ if $i \in A(\mathbf{t})$. In other words, a rule or a synchronization vector is controlled by all the LTSSs that it synchronizes.

Example 1. Let a, b, c , and d be labels, and P_1, P_2 , and P_3 be the processes defined as follows, where the initial states are those numbered 0:



Consider the network $N = ((P_1, P_2, P_3), V_{123})$, where V_{123} is the set of rules $\{((a, a, \bullet), a), ((a, \bullet, a), a), ((b, b, b), b), ((c, c, \bullet), \tau), ((\bullet, \bullet, d), d)\}$. The first two synchronization rules express that a transition labeled by a in P_1 synchronizes with a transition labeled by a nondeterministically either in P_2 or in P_3 . The third synchronization rule expresses a multiway synchronization on b between P_1, P_2 , and P_3 . The fourth synchronization rule expresses that synchronization on c between P_1 and P_2 yields a transition labeled by τ , thus is internal. The fifth synchronization rule expresses that transitions labeled by d in P_3 execute in full interleaving. The global LTS of this network is given in Figure 1.

A large set of operators can be translated to networks: An LTS P translates to a network of the form $((P), V)$ where V contains a rule of the form $((a), a)$ for each label a of P . Hiding of labels in an expression E_0 translates to the network of E_0 in which each rule (\mathbf{t}, a) with a a label to be hidden is replaced by (\mathbf{t}, τ) .

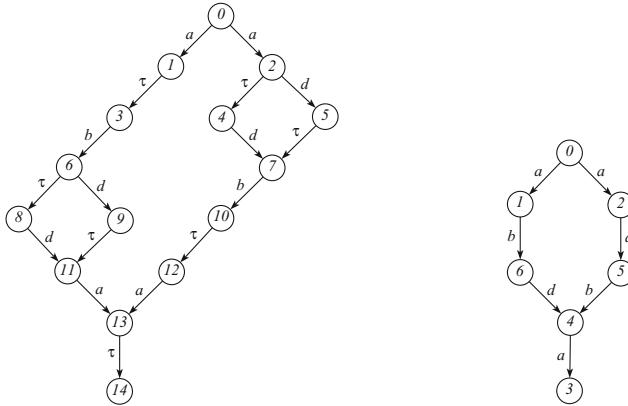


Fig. 1. Global LTS of the network of Example 1, unreduced (left) and minimized modulo branching bisimulation (right)

Renaming of labels in an expression E_0 translates to the network of E_0 in which each rule (t, a) with a a label to be renamed into a' is replaced by (t, a') . Cutting of labels in an expression E_0 translates to the network of E_0 in which each rule (t, a) with a a label to be cut is merely suppressed. Parallel composition of a set of expressions E_1, \dots, E_n translates to the network obtained by concatenating the vectors of LTSS of E_1, \dots, E_n and joining their synchronization rules as follows: for each subset $\{E_{i_1}, \dots, E_{i_m}\}$ of $\{E_1, \dots, E_n\}$ that may synchronize all together on label a , the resulting set of synchronization rules contains as many different rules of the form $(t_1 \oplus \dots \oplus t_n, a)$ as possible, such that for each $i \in \{i_1, \dots, i_m\}$ the network of E_i has a rule of the form (t_i, a) and for each $i \in 1..n \setminus \{i_1, \dots, i_m\}$, t_i is a vector of \bullet whose size is the size of the network of E_i . This translation also holds for $m = 1$, corresponding to labels that do not synchronize.

Example 2. Let P_i ($i \in 1..3$) be LTSS with labels a and b . Each P_i translates into $((P_i), \{((a), a), ((b), b)\})$. Hiding a in P_1 translates into $((P_1), \{((a), \tau), ((b), b)\})$, renaming a to c in P_1 translates into $((P_1), \{((a), c), ((b), b)\})$, cutting a in P_1 translates into $((P_1), \{((b), b)\})$, and synchronizing P_1 and P_2 on a deterministically translates into the network with vector of LTSS (P_1, P_2) and set of rules $\{((a, a), a), ((b, \bullet), b), ((\bullet, b), b)\}$. The set of rules for synchronizing P_1 and P_2 on a nondeterministically is $\{((a, a), a), ((a, \bullet), a), ((\bullet, a), a), ((b, \bullet), b), ((\bullet, b), b)\}$. Lastly, the set of rules for 2 among 3 synchronization on a between P_1 , P_2 , and P_3 is $\{((a, a, \bullet), a), ((a, \bullet, a), a), ((b, \bullet, a), a), ((\bullet, b, \bullet), b), ((\bullet, \bullet, b), b)\}$.

Rules of the form (t, a) may define synchronizations between distinct labels, which is useful, notably to represent combinations of synchronizations and renamings. For instance, the (pseudo-language) expression “**(rename** $a \rightarrow c$ **in** P_1) || (**rename** $b \rightarrow c$ **in** P_2)” (where $||$ represents synchronization on all visible labels) produces the synchronization rule $((a, b), c)$.

Many equivalence relations on LTSS exist, each preserving particular classes of properties. For instance, two LTSS that are trace equivalent have the same set of traces. Equivalences can be used to ease the cost of explicit-state verification. For instance, the traces of an LTS P can be obtained by generating a smaller LTS P' , which is trace equivalent to P . We are then interested in the smallest LTS equivalent to P . Replacing an LTS by its smallest representative with respect to an equivalence relation is called *minimizing* the LTS modulo this relation.

We are especially interested in equivalence relations that interact well with algebraic operations like parallel composition, renaming, hiding, and cutting. We say an equivalence relation R is a congruence with respect to networks if the equivalence of two LTSS P and P' implies the equivalence of any network N to a network N' obtained by replacing P by P' . Strong bisimulation and trace equivalence are congruences for networks. Branching bisimulation, observation equivalence, safety equivalence, and weak trace equivalence, are also congruences for networks provided that the synchronization rules satisfy the following standard constraints regarding the internal transitions of individual LTSS [26]:

- **No synchronization:** $(\mathbf{t}, a) \in V \wedge \mathbf{t}[i] = \tau \implies A(\mathbf{t}) = \{i\}$
- **No renaming:** $(\mathbf{t}, a) \in V \wedge \mathbf{t}[i] = \tau \implies a = \tau$
- **No cut:** $\xrightarrow{\tau}_i \neq \emptyset \implies (\exists (\mathbf{t}, \tau) \in V) \mathbf{t}[i] = \tau$

We assume that all networks in this paper satisfy these constraints.

3 Compositional Aggregation of Networks

Generating the global LTS of a network all at once may face state explosion. To overcome this, the LTS can be generated incrementally, by alternating compositions of well-chosen subsets of the individual LTSS and minimizations modulo an equivalence relation. We call *aggregation* a composition followed by a minimization of the result, and *compositional aggregation* this incremental technique.

Formally, we consider a relation R that is a congruence for networks and write $\text{min}_R(P)$ for the minimization modulo R of the LTS P . A compositional aggregation strategy to generate modulo R the LTS corresponding to a network of LTSS $N = (\mathbf{S}, V)$ of size n is defined by the following iterative algorithm:

1. Replace in N each $\mathbf{S}[i]$ ($i \in 1..n$) by $\text{min}_R(\mathbf{S}[i])$.
2. Select a set I containing at least two of the individual LTSS of N . A strategy for selection will be addressed in the next section.
3. Replace N by a new network $\text{agg}(N, I)$ — defined below — corresponding to N in which the LTSS in I have been replaced by their aggregation.
4. If N still contains more than two LTSS, then continue in step 2. Otherwise return $\text{min}_R(\text{lts}(N))$.

By abuse of language, since I denotes the set of LTSS to be aggregated, we call I an aggregation. We represent I as a subset of $1..n$, corresponding to the indexes of the LTSS to be aggregated. We assume a function $\alpha(\mathbf{t}, a)$ that associates to

$$\text{agg}(N, I) = (\min_R(\text{lts}(\text{proj}(N, I))) :: \mathbf{S}_{|\bar{I}}, V_{\text{agg}})$$

where $\text{proj}(N, I) = (\mathbf{S}_{|I}, V_{\text{proj}})$

$$V_{\text{agg}} = \{ (a :: \mathbf{t}_{|\bar{I}}, a) \mid (\mathbf{t}, a) \in V \wedge A(\mathbf{t}) \subseteq I \} \cup \\ \{ (\alpha(\mathbf{t}, a) :: \mathbf{t}_{|\bar{I}}, a) \mid (\mathbf{t}, a) \in V \wedge \emptyset \subset (I \cap A(\mathbf{t})) \subset A(\mathbf{t}) \} \cup \\ \{ (\bullet :: \mathbf{t}_{|\bar{I}}, a) \mid (\mathbf{t}, a) \in V \wedge (I \cap A(\mathbf{t})) = \emptyset \}$$

and $V_{\text{proj}} = \{ (\mathbf{t}_{|I}, a) \mid (\mathbf{t}, a) \in V \wedge A(\mathbf{t}) \subseteq I \} \cup \\ \{ (\mathbf{t}_{|I}, \alpha(\mathbf{t}, a)) \mid (\mathbf{t}, a) \in V \wedge \emptyset \subset (I \cap A(\mathbf{t})) \subset A(\mathbf{t}) \}$

Fig. 2. Definition of $\text{agg}(N, I)$

each $(\mathbf{t}, a) \in V$ a unique label distinct from all others and define $\text{agg}(N, I)$ in Figure 2, where $\min_R(\text{lts}(\text{proj}(N, I)))$ corresponds to the aggregation of the LTss inside I and $\mathbf{S}_{|\bar{I}}$ corresponds to the LTss outside I , which are kept non-aggregated. The synchronization rules V_{proj} of the auxiliary network $\text{proj}(N, I)$ are obtained by projection of V on to I , whereas the synchronization rules V_{agg} are obtained by synchronization of the labels in V_{proj} with the projection of V on to \bar{I} . The rules of V_{proj} and V_{agg} are organized in three subsets:

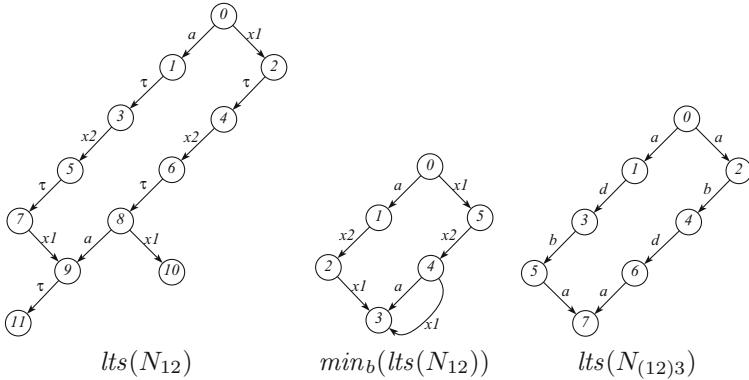
- The first subset — rules of the form $(\mathbf{t}_{|I}, a)$ in V_{proj} and $(a :: \mathbf{t}_{|\bar{I}}, a)$ in V_{agg} — represents the synchronization rules that are completely controlled by LTss inside I . In this case, $\mathbf{t}_{|\bar{I}}$ is a vector of \bullet , which expresses that transitions of $\min_R(\text{lts}(\text{proj}(N, I)))$ obtained by synchronization of LTss inside I do not need to synchronize with LTss outside I .
- The second subset — rules of the form $(\mathbf{t}_{|I}, \alpha(\mathbf{t}, a))$ in V_{proj} and $(\alpha(\mathbf{t}, a) :: \mathbf{t}_{|\bar{I}}, a)$ in V_{agg} — represents the synchronization rules that are controlled by LTss both inside I and outside I . This expresses that transitions of $\min_R(\text{lts}(\text{proj}(N, I)))$ obtained by synchronization of LTss inside I still need to synchronize with LTss outside I . The special label $\alpha(\mathbf{t}, a)$ is an intermediate label used for this synchronization. Note that in general, a cannot be used instead of $\alpha(\mathbf{t}, a)$ because (1) a can be the internal label τ (even though \mathbf{t} synchronizes only visible labels), which cannot be synchronized, and (2) in general there can be several rules with the same label a (in particular when nondeterministic synchronization is involved) and using a could create unexpected synchronizations.
- The third subset — rules of the form $(\bullet :: \mathbf{t}_{|\bar{I}}, a)$ in V_{agg} — represents the synchronization rules that are completely controlled by LTss outside I . These rules do not impose synchronization constraints on LTss inside I , thus explaining why V_{proj} has no rule in the third subset.

If P_1, \dots, P_m are individual LTss of N and if I is the set of their indexes, then we may write $\text{comp}(P_1, \dots, P_m)$ for $\text{lts}(\text{proj}(N, I))$.

Since R is a congruence, $\text{lts}(\text{agg}(N, I))$ is equivalent modulo R to $\text{lts}(N)$. Therefore, $\text{lts}(N)$ remains invariantly R -equivalent to the input until the end of the algorithm, thus guaranteeing the correctness of compositional aggregation. Moreover, the size of $\text{agg}(N, I)$ is the size of N plus 1 minus the size of I (which

is at least 2). Since N is substituted by $\text{agg}(N, I)$ at each step, this guarantees that the size of N decreases and therefore that the algorithm terminates.

Example 3. We write $\min_b(P)$ for minimization of P modulo branching bisimulation. The global LTS of the network of Example 1, whose LTSS P_1 , P_2 , and P_3 are already minimal, can be generated modulo branching bisimulation by first composing P_1 and P_2 , then P_3 as follows. First, build $N_{12} = \text{proj}(N, \{1, 2\}) = ((P_1, P_2), V_{12})$ for the aggregation of P_1 and P_2 , where V_{12} is the set of rules $\{((a, a), a), ((a, \bullet), \mathbf{x}_1), ((b, b), \mathbf{x}_2), ((c, c), \tau)\}$, \mathbf{x}_1 is the label $\alpha((a, \bullet, a), a)$, and \mathbf{x}_2 is the label $\alpha((b, b, b), b)$. Compute the intermediate LTS $P_{12} = \min_b(\text{lts}(N_{12}))$ (see below). Second, build $N_{(12)3} = \text{agg}(N, \{1, 2\}) = ((P_{12}, P_3), V_{(12)3})$, where $V_{(12)3}$ is the set $\{((a, \bullet), a), ((\mathbf{x}_1, a), a), ((\mathbf{x}_2, b), b), ((\tau, \bullet), \tau), ((\bullet, d), d)\}$. Return $\min_b(\text{lts}(N_{(12)3}))$ (see $\text{lts}(N_{(12)3})$ below and $\min_b(\text{lts}(N_{(12)3}))$ in Figure 1, right), which is branching equivalent to $\text{lts}(N_{123})$ (see Figure 1, left). Note that both LTSS $\text{lts}(N_{12})$ and $\text{lts}(N_{(12)3})$ are smaller than $\text{lts}(N_{123})$.



Other aggregation orders are possible, yielding different intermediate graphs. Figure 3 gives the sizes of those intermediate graphs corresponding to the different aggregation orders. The largest intermediate graph size of each order is indicated in bold type. This table shows that the aggregation order described above (order 1) is optimal in terms of largest intermediate graph (12 transitions).

Note that N has the same meaning as the LOTOS composition of processes “**hide** c in $(P_1 \parallel [a, b, c] \parallel (P_2 \parallel [b] \parallel P_3))$ ”. Yet using LOTOS operators instead of networks, it would not be possible to aggregate P_1 , P_2 , and P_3 following the optimal order (P_1, P_2) then P_3 , because there do not exist LOTOS operators (or compositions of operators) op_1 and op_2 , such that the above term can be written in the form “ $op_2(op_1(P_1, P_2), P_3)$ ”. For instance, “**hide** c in $((P_1 \parallel [a, b, c] \parallel P_2) \parallel [b] \parallel P_3)$ ” does not work, because we lose synchronization on a between P_1 and P_3 . More generally, the problem happens when the model contains nondeterministic synchronizations, which can make parallel composition non-associative. Similar examples can be built in other process algebras such as, e.g., CSP (by combining renaming and synchronization on common alphabet).

Order 1 : (P_1, P_2) then P_3	states	transitions
$comp(P_1, P_2)$	12	12
$min_b(comp(P_1, P_2))$	6	7
$comp(min_b(comp(P_1, P_2)), P_3)$	8	8
$min_b(comp(min_b(comp(P_1, P_2)), P_3))$	7	7
Order 2 : (P_1, P_3) then P_2	states	transitions
$comp(P_1, P_3)$	19	23
$min_b(comp(P_1, P_3))$	17	22
$comp(min_b(comp(P_1, P_3)), P_2)$	15	17
$min_b(comp(min_b(comp(P_1, P_3)), P_2))$	7	7
Order 3 : (P_2, P_3) then P_1	states	transitions
$comp(P_2, P_3)$	18	34
$min_b(comp(P_2, P_3))$	18	34
$comp(min_b(comp(P_2, P_3)), P_1)$	15	17
$min_b(comp(min_b(comp(P_2, P_3)), P_1))$	7	7
Order 4 : (P_1, P_2, P_3)	states	transitions
$comp(P_1, P_2, P_3)$	15	17
$min_b(comp(P_1, P_2, P_3))$	7	7

Fig. 3. Sizes of intermediate graphs for all aggregation orders

4 Smart Reduction

The most difficult issue in compositional aggregation concerns step 2 of the algorithm, namely to select, if possible automatically, an aggregation I that avoids state explosion. In this section, we present *smart reduction*, which corresponds to compositional aggregation using a heuristic based on metrics evaluated against possible aggregations.

Our metrics use an estimate of the number of global transitions in I generated by synchronization vector \mathbf{t} , written $ET(I, \mathbf{t})$ and defined below:

$$ET(I, \mathbf{t}) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } I \cap A(\mathbf{t}) = \emptyset \\ \prod_{i \in I \setminus A(\mathbf{t})} |\Sigma_i| \times \prod_{i \in I \cap A(\mathbf{t})} |\xrightarrow{\mathbf{t}[i]}_i| & \text{otherwise} \end{cases}$$

Informally, $ET(I, \mathbf{t})$ counts, for vector \mathbf{t} , the number of transitions going out of every product state of I , including unreachable states. This count equals 0 if \mathbf{t} is not controlled by Ltss inside I (first line). Otherwise, $proj(N, I)$ has a rule of the form $(\mathbf{t}|_I, b)$. This rule generates a transition in a state of $proj(N, I)$ without condition on the states of the individual Ltss $\mathbf{S}[i]$ such that $i \in I \setminus A(\mathbf{t})$ — thus justifying the first product in the definition of $ET(I, \mathbf{t})$ — and provided the states of the individual Ltss $\mathbf{S}[i]$ such that $i \in I \cap A(\mathbf{t})$ have a transition labeled $\mathbf{t}[i]$ — thus justifying the second product. In general, the exact number of global transitions in I generated by synchronization vector \mathbf{t} is below this count since some states may be unreachable.

We now define our metrics on networks. The *hiding metric* is defined by $HM(I) \stackrel{\text{def}}{=} HR(I)/|I|$, where $HR(I)$ (the *hiding rate*) is defined in Figure 4 (left). Informally, $HR(I)$ represents an estimate of the proportion of transitions in $\text{proj}(N, I)$ that are internal. Those transitions are necessarily created by rules completely controlled by LTSS inside I . The addition of 1 in its divisor avoids division by 0 in pathological cases. The divisor $|I|$ of $HM(I)$ aims at favouring smaller aggregations, which are likely to yield smaller intermediate LTSS.

Using $HM(I)$ is justified in the context of weak equivalence relations (e.g., branching bisimulation), because internal transitions are often eliminated by the corresponding minimizations. Although to a lesser extent, it is also justified in the context of strong bisimulation, because hiding enables abstracting away from labels that otherwise would differentiate the behaviour of equivalent states. However, in both cases, $HM(I)$ is not sufficient to avoid intermediate explosion due to the aggregation of loosely synchronized LTSS. To palliate this, the hiding metric will be combined with the *interleaving metric* $IM(I) \stackrel{\text{def}}{=} (1 - IR(I))/|I|$, where the *interleaving rate* $IR(I)$ is defined in Figure 4 (right). In this definition, $\mathbf{t}@i$ denotes the synchronization vector of size n defined by $(\mathbf{t}@i)[i] = \mathbf{t}[i]$ and $(\forall j \in 1..n \setminus \{i\}) (\mathbf{t}@i)[j] = \bullet$. The value $IR(I)$ is therefore the quotient between an estimate of the number of global transitions of I and the number of global transitions that I would have if all individual LTSS were fully interleaving. For an aggregation I of fully interleaving individual LTSS, we thus have $IR(I)$ very close to 1. It is a refinement of the *interleaving count* defined in [11], which uses the proportion of fully interleaving (i.e., non-synchronized) individual LTS transitions out of the total number of individual LTS transitions. We believe that $IR(I)$ is more accurate, because it also measures the *partial* interleaving of synchronized transitions with the remainder of the aggregation. Taking into account both the hiding rate and the interleaving rate, we use here the *combined metric* $CM(I) \stackrel{\text{def}}{=} HM(I) + IM(I)$.

$$HR(I) \stackrel{\text{def}}{=} \frac{\sum_{(\mathbf{t}, \tau) \in V \wedge A(\mathbf{t}) \subseteq I} ET(I, \mathbf{t})}{1 + \sum_{(\mathbf{t}, a) \in V} ET(I, \mathbf{t})} \quad IR(I) \stackrel{\text{def}}{=} \frac{\sum_{(\mathbf{t}, a) \in V} ET(I, \mathbf{t})}{1 + \sum_{(\mathbf{t}, a) \in V} \sum_{i \in I \cap A(\mathbf{t})} ET(I, \mathbf{t}@i)}$$

Fig. 4. Hiding rate and interleaving rate of an aggregation I

Smart reduction selects the aggregation to which the metrics gives the highest value (high proportion of internal transitions and low interleaving). To avoid the combinatorial explosion of the number of aggregations, we proceed as in [11] and only consider: (1) aggregations whose size is bounded by a constant (definable by the user), and (2) aggregations that are *connected*. An aggregation is connected if for each pair of distinct LTSS P_i, P_j in the aggregation, P_i and P_j are *connected*, which is defined recursively as follows: either there is a synchronization rule (\mathbf{t}, a) such that $\{i, j\} \subseteq A(\mathbf{t})$ (i.e., P_i and P_j are synchronized) or, recursively, the aggregation contains a third (distinct) LTS P_k connected to both P_i and P_j .

Example 4. The metrics evaluate as follows on the network of Examples 1 and 3:

I	$\{1, 2\}$	$\{1, 3\}$	$\{2, 3\}$	$\{1, 2, 3\}$
$HM(I)$	0.211	0	0	0.129
$IM(I)$	0.357	0.227	0.124	0.249
$CM(I)$	0.568	0.227	0.124	0.378

As expected, the combined metric gives the highest value to $\{1, 2\}$, therefore designating it as the best aggregation in the first step. Note that, more generally in this example, a comparison between the graph sizes in Figure 3 and the values in the above table shows that one aggregation is more efficient than the others whenever the combined metrics gives it a higher value.

5 Implementation

Smart reduction has been implemented in CADP (*Construction and Analysis of Distributed Processes*)¹ [17], a widely disseminated toolbox for the design of communication protocols and distributed systems. CADP offers a large set of features, ranging from step-by-step simulation to massively parallel model-checking. It is the only toolbox to offer compilers for several input formalisms (LOTOS, LOTOS NT, networks of automata, etc.) into LTSS, equivalence checking tools (minimization and comparisons modulo bisimulation relations), model-checkers for various temporal logics and μ -calculus, several verification techniques combined together (enumerative verification, on-the-fly verification, compositional aggregation, partial order reduction, distributed model checking, etc.), and a number of other tools providing advanced functionalities such as visual checking, performance evaluation, etc. The tools SVL and EXP.OPEN 2.0 have been extended to support smart reduction.

The tool SVL. SVL (*Script Verification Language*) [16] is both a scripting language that enables advanced verification scenarios to be described at a high-level of abstraction, and an associated compiler that enables automatic execution of the SVL scripts. Smart reduction is available as a new SVL operator, which can be parameterized by an equivalence relation. For example, the following script describes the smart branching reduction of the LOTOS behaviour corresponding to the network of Example 1:

```
% DEFAULT_LOTOS_FILE="processes.lotos"
% DEFAULT_SMART_LIMIT=3
"composition.bcg" = smart branching reduction of
    hide c in (P1 |[ a, b, c ]| (P2 |[ b ]| P3));
```

The file “processes.lotos” is where the three processes P1, P2 and P3 are specified, and “composition.bcg” is the name of the file where the LTS resulting from their aggregation is to be stored, represented in a compact graph

¹ <http://vasy.inria.fr/cadp>

format called BCG (*Binary Coded Graph*). The (optional) line of the form “%
DEFAULT_SMART_LIMIT=3” defines as 3 the maximal number of LTss aggregated at each step. Otherwise, a default value of 4 is used in our implementation. This script aggregates the three processes in the order determined automatically by the tool, which in this case is the optimal order, P1 and P2, then P3.

The tool EXP.OPEN 2.0. EXP.OPEN 2.0 [26] is a conservative extension of the former version 1.0, developed in 1995 by L. Mounier (Univ. Joseph Fourier, Grenoble, France). It takes as input an expression consisting of LTss composed together using the parallel composition, label hiding, label renaming, and label cutting operators of the process algebras LOTOS [24], CCS [27], CSP [23], and μ CRL [21], as well as the generalized parallel composition operator of E-LOTOS and LOTOS NT [18], and synchronization vectors in the style of MEC [1] and FC2 [3]. This expression is compiled into a network. In standard usage, the network is compiled into an implicit representation in C of the global LTS (initial state, transition function, etc.), which can be linked to various application programs available in CADP for simulation or verification purposes, following the OPEN/CÆSAR architecture [14]. In the framework of smart reduction, EXP.OPEN is invoked by SVL for computing the metrics, generating the networks $proj(N, I)$ and $agg(N, I)$, and generating the corresponding global LTss.

6 Experimental Results

We have applied smart branching reduction to a set of case-studies and compared it with two other compositional aggregation strategies already implemented in SVL, namely *root leaf reduction*, which consists in aggregating all individual LTss at once, and *node reduction*, which consists in aggregating the individual LTss one after the other in a syntactical order given by the term describing the composition. The results are given in Figure 5, which provides for each strategy the largest number of transitions in the generated intermediate LTSs. The strategy named “Smart (HM)” (resp. “Smart (IM)”) corresponds to the hiding (resp. interleaving) metric, whereas “Smart (CM)” corresponds to the combined metric. The smallest number of each line is written in bold type.

These experiments show that smart reduction is very often better than, and generally comparably efficient to, root leaf reduction and node reduction. Notable exceptions are the CFS and DFT IL experiments, for which root leaf reduction is noticeably more efficient. One reason is that EXP.OPEN uses partial order reductions and composing all individual LTss at once may enable partial order reductions that cannot be applied to partial aggregations.

Although both hiding and interleaving metrics used separately may, in a few cases, yield slightly better results than the combined metric, we did not see cases where the combined metric is far worse than all other strategies, like the hiding metric is for TN and the interleaving metric for CFS. In that sense, combining both hiding and interleaving metrics seems to make the heuristic more robust.

Experiment	Node	Root leaf	Smart (HM)	Smart (IM)	Smart (CM)
ABP 1	380	328	210	104	104
ABP 2	2, 540	2, 200	1, 354	504	504
Cache	1, 925	1, 925	1, 848	1, 925	1, 925
CFS	2, 193, 750	486, 990	1, 366, 968	96, 040, 412	5, 113, 256
DES	22, 544	3, 508	3, 508	14, 205	14, 205
DFT CAS	95, 392	99, 133	336	346	346
DFT HCPS	4, 730	79, 509	425	435	435
DFT IL	29, 808	316	2, 658	1, 456	2, 658
DFT MDCS	635, 235	117, 772	536	5, 305	346
DFT NDPS	17, 011	1, 857	393	449	346
DLE 1	15, 234	7, 660	7, 709	10, 424	9, 883
DLE 2	8, 169	2, 809	2, 150	1, 852	2, 150
DLE 3	253, 272	217, 800	181, 320	231, 616	175, 072
DLE 4	33, 920	29, 584	25, 008	8, 896	26, 864
DLE 5	1, 796, 616	1, 796, 616	1, 403, 936	1, 716, 136	1, 433, 640
DLE 6	35, 328	35, 328	5, 328	5, 328	5, 328
DLE 7	612, 637	486, 491	369, 810	583, 289	577, 775
HAVi async	145, 321	22, 703	21, 645	21, 862	21, 809
HAVi sync	19, 339	5, 021	4, 743	4, 743	4, 743
NFP	199, 728	1, 986, 768	104, 960	89, 696	89, 696
ODP	158, 318	158, 318	87, 936	39, 841	87, 936
RelRel 1	28, 068	9, 228	9, 282	5, 574	5, 574
RelRel 2	11, 610, 235	5, 341, 821	5, 341, 821	5, 341, 821	5, 341, 821
SD 1	21, 870	3, 482	19, 679	4, 690	19, 679
SD 2	6, 561	11, 997	3, 624	2, 297	3, 192
SD 3	1, 944	32, 168	1, 380	896	1, 164
SD 4	633, 130	1, 208, 592	975, 872	789, 886	975, 872
TN	54, 906, 000	746, 880	69, 547, 712	749, 312	709, 504

Fig. 5. Experimental results

7 Conclusion

We have presented smart reduction, an automated compositional aggregation strategy used to generate modulo an equivalence relation the LTS of a system made of processes composed in parallel, which generalizes previous work [29,11]. It uses a heuristic based on a metric for evaluating the potential for an aggregation to avoid state explosion. The metric applies to the expressive network model, used internally by the EXP.OPEN tool, thus enabling the application of smart reduction to a large variety of operators (including synchronization vectors and operators from LOTOS, LOTOS NT, E-LOTOS, Ccs, CSP, and μ CRL), while avoiding the language restrictions that could otherwise make the optimal order unavailable. We have provided an implementation and experimentation of smart reduction in the framework of the CADP toolbox. The resulting

strategy yields good results, often better than systematic strategies such as node reduction (aggregating LTss one after the other in an order given by the term describing the composition) and root leaf reduction (aggregating all LTss at once). Most importantly, the metric that combines both the hiding and the interleaving metrics is *robust* in the sense that (for the test cases considered) it never leads to extremely bad orders. Smart reduction is well-integrated in the framework of the SVL scripting language of CADP, thus making it very easy to use. Smart reduction allows users of CADP to enjoy the benefits of compositional aggregation without having to guess or experimentally find a good composition order. Moreover, it enables the automatic verification of software systems that rely on elaborate forms of compositions, by combining automatic translations to compositional models and automatic compositional aggregation. Crucially, such verification can be used by software engineers or architects who have no background in formal methods. In future, we would like to have the metric integrate information about the amount of partial order reduction that can be expected in each aggregation, so as to provide even better aggregation strategies.

Acknowledgements. The authors are grateful to Christine McKinty, Gwen Salaün, Damien Thivolle, and Verena Wolf for their useful comments on this paper.

References

1. Arnold, A.: MEC: A System for Constructing and Analysing Transition Systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, Springer, Heidelberg (1990)
2. Berthomieu, B., Bodeveix, J.-P., Farail, P., Filali, M., Garavel, H., Gaufillet, P., Lang, F., Vernadat, F.: FIACRE: an Intermediate Language for Model Verification in the TOPCASED Environment. In: Proc. of ERTS (2008)
3. Bouali, A., Ressouche, A., Roy, V., de Simone, R.: The Fc2Tools set: a Toolset for the Verification of Concurrent Systems. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, Springer, Heidelberg (1996)
4. Boudali, H., Crouzen, P., Stoelinga, M.: Compositional Semantics for Dynamic Fault Trees in Terms of Interactive Markov Chains. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 441–456. Springer, Heidelberg (2007)
5. Boudali, H., Crouzen, P., Stoelinga, M.: Dynamic fault tree analysis using input/output interactive Markov chains. In: Proc. of Dependable Systems and Networks. IEEE, Los Alamitos (2007)
6. Boudali, H., Crouzen, P., Haverkort, B.R., Kuntz, M., Stoelinga, M.: Architectural dependability evaluation with Arcade. In: Proc. of Dependable Systems and Networks. IEEE, Los Alamitos (2008)
7. Boudali, H., Sözer, H., Stoelinga, M.: Architectural Availability Analysis of Software Decomposition for Local Recovery. In: Proc. of Secure Software Integration and Reliability Improvement (2009)
8. Chehaibar, G., Garavel, H., Mounier, L., Tawbi, N., Zulian, F.: Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In: Proc. of FORTE/PSTV. Chapman and Hall, Boca Raton (1996)

9. Cheung, S.C., Kramer, J.: Enhancing Compositional Reachability Analysis with Context Constraints. In: Proc. of ACM SIGSOFT International Symposium on the Foundations of Software Engineering. ACM Press, New York (1993)
10. Coste, N., Garavel, H., Hermanns, H., Hersemeule, R., Thonnart, Y., Zidouni, M.: Quantitative Evaluation in Embedded System Design: Validation of Multiprocessor Multithreaded Architectures. In: Proc. of DATE (2008)
11. Crouzen, P., Hermanns, H.: Aggregation Ordering for Massively Parallel Compositional Models. In: Proc. of ACSD. IEEE, Los Alamitos (2010)
12. Cubo, J., Salaün, G., Canal, C., Pimentel, E., Poizat, P.: A Model-Based Approach to the Verification and Adaptation of WF/.NET Components. ENTCS 215 (2008)
13. Foster, H., Uchitel, S., Magee, J., Kramer, J.: LTSA-WS: a tool for model-based verification of web service compositions and choreography. In: Proc. of ICSE (2006)
14. Garavel, H.: OPEN/CAESAR: An Open Software Architecture for Verification, Simulation, and Testing. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, p. 68. Springer, Heidelberg (1998)
15. Garavel, H., Hermanns, H.: On Combining Functional Verification and Performance Evaluation Using CADP. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, p. 410. Springer, Heidelberg (2002)
16. Garavel, H., Lang, F.: SVL: a Scripting Language for Compositional Verification. In: Proc. of FORTE. Kluwer, Dordrecht (2001)
17. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
18. Garavel, H., Sighireanu, M.: A Graphical Parallel Composition Operator for Process Algebras. In: Proc. of FORTE/PSTV. Kluwer, Dordrecht (1999)
19. Giannakopoulou, D., Kramer, J., Cheung, S.C.: Analysing the behaviour of distributed systems using TRACTA. Journal of Automated Software Engineering 6(1), 7–35 (1999)
20. Graf, S., Steffen, B., Lütten, G.: Compositional Minimization of Finite State Systems using Interface Specifications. Formal Aspects of Computation 8(5), 607–616 (1996)
21. Groote, J.F., Ponse, A.: The Syntax and Semantics of μ CRL. In: Proc. of ACP. Workshops in Computing Series (1995)
22. Hermanns, H., Katoen, J.-P.: Automated Compositional Markov Chain Generation for a Plain-Old Telephone System. Science of Computer Programming 36, 97–127 (2000)
23. Hoare, C.A.R.: Communicating Sequential Processes. Communications of the ACM 21(8), 666–677 (1978)
24. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization, Genève (1989)
25. Krimm, J.-P., Mounier, L.: Compositional State Space Generation from LOTOS Programs. In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, Springer, Heidelberg (1997)
26. Lang, F.: Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-The-Fly Verification Methods. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 70–88. Springer, Heidelberg (2005)

27. Milner, R.: A Calculus of Communicating Systems. In: Milner, R. (ed.) A Calculus of Communication Systems. LNCS, vol. 92, Springer, Heidelberg (1980)
28. Rosa, N.S., Cunha, P.R.F.: A LOTOS Framework for Middleware Specification. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 136–142. Springer, Heidelberg (2006)
29. Tai, K.C., Koppol, V.: Hierarchy-Based Incremental Reachability Analysis of Communication Protocols. In: Proc. of Network Protocols. IEEE, Los Alamitos (1993)
30. Scollo, G., Zecchini, S.: Architectural Unit Testing. ENTCS 111 (2005)
31. Tronel, F., Lang, F., Garavel, H.: Compositional Verification Using CADP of the ScalAgent Deployment Protocol for Software Components. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 244–260. Springer, Heidelberg (2003)
32. Valmari, A.: Compositional State Space Generation. In: Rozenberg, G. (ed.) APN 1993. LNCS, vol. 674, Springer, Heidelberg (1993)

Uniform Monte-Carlo Model Checking

Johan Oudinet^{1,2}, Alain Denise^{1,2,3}, Marie-Claude Gaudel^{1,2},
Richard Lassaigne^{4,5}, and Sylvain Peyronnet^{1,2,3}

¹ Univ Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405

² CNRS, Orsay, F-91405

³ INRIA Saclay - Île-de-France, F-91893 Orsay Cedex

⁴ Univ. Paris VII, Equipe de Logique Mathématique, UMR7056

⁵ CNRS, Paris-Centre, F-75000

Abstract. Grosu and Smolka have proposed a randomised Monte-Carlo algorithm for LTL model-checking. Their method is based on random exploration of the intersection of the model and of the Büchi automaton that represents the property to be checked. The targets of this exploration are so-called *lassos*, i.e. elementary paths followed by elementary circuits. During this exploration outgoing transitions are chosen uniformly at random.

Grosu and Smolka note that, depending on the topology, the uniform choice of outgoing transitions may lead to very low probabilities of some lassos. In such cases, very big numbers of random walks are required to reach an acceptable coverage of lassos, and thus a good probability either of satisfaction of the property or of discovery of a counter-example. In this paper, we propose an alternative sampling strategy for lassos in the line of the uniform exploration of models presented in some previous work.

The problem of finding all elementary cycles in a directed graph is known to be difficult: there is no hope for a polynomial time algorithm. Therefore, we consider a well-known sub-class of directed graphs, namely the reducible flow graphs, which correspond to well-structured programs and most control-command systems.

We propose an efficient algorithm for counting and generating uniformly lassos in reducible flowgraphs. This algorithm has been implemented and experimented on a pathological example. We compare the lasso coverages obtained with our new uniform method and with uniform choice among the outgoing transitions.

1 Introduction

Random exploration of large models is one of the ways of fighting the state explosion problem. In [11], Grosu and Smolka have proposed a randomized Monte-Carlo algorithm for LTL model-checking together with an implementation called *MC²*. Given a finite model M and an LTL formula Φ , their algorithm performs random walks ending by a cycle, (the resulting paths are called *lassos*) in the Büchi automaton $B = B_M \times B_{\neg \Phi}$ to decide whether $L(B) = \emptyset$ with a probability which depends on the number of performed random walks. More precisely,

their algorithm samples lassos in the automaton B until an *accepting* lasso is found or a fixed bound on the number of sampled lassos is reached. If MC^2 find an accepting lasso, it means that the target property is false (by construction of B , an accepting lasso is a counterexample to the property). On the contrary, if the algorithm stops without finding an accepting lasso, then the probability that the formula is true is high. The advantage of a tool such as MC^2 is that it is fast, memory-efficient, and scalable.

Random exploration of a model is a classical approach in simulation [3] and testing (see for instance [26,6]) and more recently in model-checking [21,8,20,1]. A usual way to explore a model at random is to use isotropic random walks: given the states of the model and their successors, an isotropic random walk is a succession of states where at each step, the next state is drawn uniformly at random among the successors, or, as in [11] the next transition is drawn uniformly at random among the outgoing transitions. This approach is easy to implement and only requires local knowledge of the model.

However, as noted in [21] and [19], isotropic exploration may lead to bad coverage of the model in case of irregular topology of the underlying transition graph. Moreover, for the same reason, it is generally not possible to get any estimation of the coverage obtained after one or several random walks: it would require some complex global analysis of the topology of the model.

Not surprisingly, it is also the case when trying to cover lassos: Figure 1 from [11], shows a Büchi automaton with $q + 1$ lassos: l_0, l_1, \dots, l_q where l_i is the lasso $s_0 s_1 \dots s_i s_0$. With an isotropic random walk, l_q has probability $1/2^q$ to be traversed.

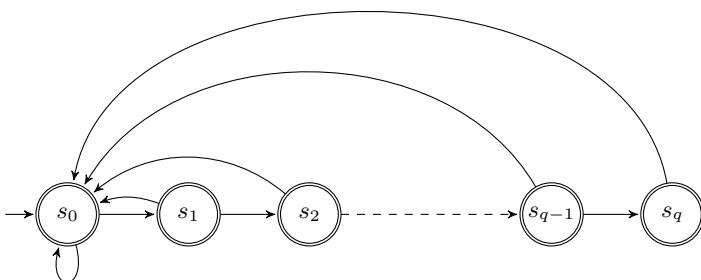


Fig. 1. A pathological example for Büchi automata

In this paper, we propose an alternative sampling strategy for lassos in the line of the uniform exploration of models presented in [7,17] and [10]. In the example above, the low probability of l_q comes from the choice done by the random walk at each state. It has to choose between a state that leads to a single lasso and a state that leads to an exponential number of lassos. In the case of an isotropic random walk, those two states have the same probability. If the number of lassos that start from each state is known, the choice of the successors can be guided to balance the probability of lassos so as to get a uniform distribution and to avoid

lassos with a too small probability. Coming back to Figure 1, with a uniform distribution on lassos, l_q has probability $1/q$ to be traversed instead of $1/2^q$.

However, the problem of counting and finding all elementary cycles in a directed graph is known to be difficult. We briefly recall in Section 2 why there is no hope of polynomial time algorithms for this problem. Therefore, we consider a well-known sub-class of directed graphs, namely the reducible flow graphs [12], which correspond to well-structured programs and most control-command systems. In Section 3, we show that the set of lassos in such graphs is exactly the set of paths that start from the initial state and that end just after the first back edge encountered during a depth-first search. Then on the basis of the methods for counting and generating paths uniformly at random, presented in [7,17] and [10], we give an algorithm for counting the number of lassos in a reducible flowgraph and uniformly generating random lassos in reducible flowgraphs. Section 4 presents how this algorithm can be used for LTL model-checking. Section 5 reports how this algorithm has been implemented and how it has been experimented on an example similar to the pathological example in Figure 1. We compare the lasso coverages obtained with our new uniform method and with uniform choice among the successors or the outgoing transitions.

2 Counting and Generating Lassos in Directed Graphs

A lasso in a graph is a finite path followed by an elementary, or simple, cycle. There are two enumeration problems for elementary cycles in a graph. The first one is counting and the second one is finding all such cycles. These two problems are hard to compute: let FP be the class of functions computable by a Turing machine running in polynomial time and $\#\text{CYCLE}(G)$ be the number of elementary cycles in a graph G ; one can prove that $\#\text{CYCLE}(G) \in FP$ implies $P = NP$ by reducing the Hamiltonian circuit problem to decide if the number of elementary cycles in a graph is large.

For the problem of finding all elementary cycles in a directed graph there is no hope for a polynomial time algorithm. For example, the number of elementary cycles in a complete directed graph grow faster than the exponential of the number of vertices. Several algorithms were designed for the finding problem. In the algorithms of Tiernan [23] and Weinblatt [25] time exponential in the size of the graph may elapse between the output of a cycle and the next. However, one can obtain enumeration algorithms with a polynomial delay between the output of two consecutive cycles.

Let G be a graph with n vertices, e edges and c elementary cycles. Tarjan [22] presented a variation of Tiernan's algorithm in which at most $O(n \cdot e)$ time elapses between the output of two cycles in sequence, giving a bound of $O(n \cdot e(c+1))$ for the running time of the algorithm. To our knowledge, the best algorithm for the finding problem is Johnson's [14], in which time consumed between the output of two consecutive cycles as well as before the first and after the last cycle never exceeds the size of the graph, resulting in bounds $O((n + e) \cdot (c+1))$ for time and $O(n + e)$ for space.

Because of the complexity of these problems in general graphs, we consider in this paper a well-known sub-class of directed graphs, namely the reducible flow graphs [12]. Control graphs of well-structured programs are reducible. Most data-flow analysis algorithms assume that the analysed programs satisfy this property, plus the fact that there is a unique final vertex reachable from any other vertex. Similarly, well-structured control-command systems correspond to reducible dataflow graphs. But in their case, any vertex is considered as final: this makes it possible the generalisation of data-flow analysis and slicing techniques to such systems [15]. Informally, the requirement on reducible graphs is that any cycle has a unique entry vertex.

It means that a large class of critical systems correspond to such flowgraphs. However, arbitrary multi-threaded programs don't. But in many cases where there are some constraints on synchronisations, for instance in cycle-driven systems, reducibility is satisfied.

The precise definition of reducibility is given below, as well as a method for counting and uniformly generating lassos in such graphs.

3 Uniform Random Generation of Lassos in Reducible Flowgraphs

A *flowgraph* $G = (V, E)$ is a graph where any vertex of G is reachable from a particular vertex of the graph called the source (we denote this vertex by s in the following). From a flowgraph G one can extract a spanning subgraph (e.g. a directed rooted tree whose vertex set is also V) with s as root. This spanning subgraph is known as *directed rooted spanning tree* (DRST). In the specific case where a depth-first search on the flowgraph and its DRST lead to the same order over the set of vertices, we call the DRST a *depth-first search tree* (DFST). The set of back edges is denoted by B_E . Given a DFST of G , we call *back edge* any edge of G that goes from a vertex to one of its ancestors in the DFST. And we say that a vertex u *dominates* a vertex v if every path from s to v in G crosses u .

Here we focus on *reducible flowgraphs*. The intuition for reducible flowgraphs is that any loop of such a flowgraph has a unique entry, that is a unique edge from a vertex exterior to the loop to a vertex in the loop. This notion has been extensively studied (see for instance [12]). The following proposition summarizes equivalent definitions of reducible flowgraphs.

Proposition 1. *All the following items are equivalent:*

1. $G = (V, E, s)$ is a reducible flowgraph.
2. Every DFS on G starting at s determines the same back edge set.
3. The directed acyclic graph (DAG) $\text{dag}(G) = (V, E - B_E, s)$ is unique.
4. For every $(u, v) \in B_E$, v dominates u .
5. Every cycle of G has a vertex which dominates the other vertices of the cycle.

We now recall the precise definition of lassos:

Definition 1 (lassos). Given a flowgraph $G = (V, E, s)$, a path in G is a final sequence of vertices s_0, s_1, \dots, s_n such that $s_0 = s$ and $\forall 0 \leq i < n \quad (s_i, s_{i+1}) \in E$. An elementary path is a path such that the vertices are pairwise distinct. A lasso is a path such that s_0, \dots, s_{n-1} is an elementary path and $s_n = s_i$ for some $0 \leq i < n$.

We can now state our first result on lassos in reducible flowgraphs.

Proposition 2. Any lasso of a reducible flowgraph ends with a back edge, and any back edge is the last edge of a lasso.

Proof. Suppose a traversal starting from the source vertex goes through a lasso and finds a back edge before the end of the lasso. Since the graph is a reducible flowgraph this means that we have a domination, thus we see two times the same vertex in the lasso, which is a contradiction with the fact that a lasso is elementary (remember a lasso is an elementary path).

Suppose now that the traversal never sees a back edge. Then no vertex has been seen twice, thus the traversal is not a lasso.

Using this proposition, we can now design a simple algorithm for

- counting the number of lassos in a reducible flowgraph,
- uniformly generating random lassos in a reducible flowgraph.

Here uniformly means equiprobably. In other word any lasso has the same probability to be generated as the others.

Following the previous proposition, the set of lassos is exactly the set of paths that start from the source and that end just after the first back edge encountered. At first, we change the problem of generating lassos into a problem of generating paths of a given size n , by slightly changing the graph: we add a new vertex s_0 with a loop, that is an edge from itself to itself, and edge from s_0 to s . And we give n a value that is an upper bound of the length of the lassos in the new graph. A straightforward such value is the depth of the depth-first search tree plus one.

Now for any vertex u , let us denote $f_u(k)$ the number of paths of length k starting from vertex u and finish just after the first back edge encountered. The edge (s_0, s_0) is not considered as a back edge because it does not finish a lasso in the initial graph G . The number of lassos is given by $f_{s_0}(n)$. And we have the following recurrence:

- $f_{s_0}(1) = 0$.
- $f_u(1) = \text{number of back edges from } u \text{ if } u \neq s_0$.
- $f_u(k) = \sum_{(u,v) \in E'} f_v(k-1)$ for $k > 1$ where E' is the set of edges of the graph, except the back edges.

The principle of the generation process is simple: starting from s , draw a path step by step. At each step, the process consists in choosing a successor of the current vertex and going to it. The problem is to proceed in such a way that

only (and all) lassos of length n can be generated, and that they are equiprobably distributed. This is done by choosing successors with suitable probabilities. Suppose that, at one given step of the generation, we are on state u , which has k successors denoted v_1, v_2, \dots, v_k . In addition, suppose that $m > 0$ transitions remain to be crossed in order to get a lasso of length n . Then the condition for uniformity is that the probability of choosing state v_i ($1 \leq i \leq k$) equals $f_{v_i}(m-1)/f_u(m)$. In other words, the probability to go to any successor of u must be proportional to the number of lassos of suitable length from this successor.

Computing the numbers $f_u(i)$ for any $0 \leq i \leq n$ and any state u of the graph can be done by using the recurrence rules above.

Table 1 presents the recurrence rules which correspond to the automaton of Figure 2.

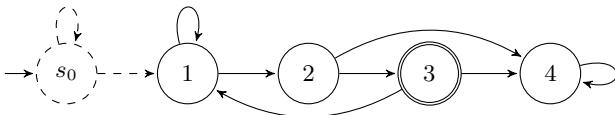


Fig. 2. An example of a Büchi automaton, from [11]. We have added the vertex s_0 and its incident edges, according to our procedure for generating lassos of length $\leq n$ for any given n .

Table 1. Recurrences for the $f_i(k)$

$$\begin{aligned}
 f_{s_0}(1) &= f_2(1) = 0 \\
 f_1(1) &= f_3(1) = f_4(1) = 1 \\
 f_{s_0}(k) &= f_{s_0}(k-1) + f_1(k-1) \quad (k > 1) \\
 f_1(k) &= f_2(k-1) \quad (k > 1) \\
 f_2(k) &= f_3(k-1) + f_4(k-1) \quad (k > 1) \\
 f_3(k) &= f_4(k-1) \quad (k > 1) \\
 f_4(k) &= 0 \quad (k > 1)
 \end{aligned}$$

Now a primitive generation scheme is as follows:

- Preprocessing stage: Compute a table of the $f_u(i)$'s for all $0 \leq i \leq n$ and any state u .
- Generation stage: Draw the lassos according to the scheme seen above.

Note that the preprocessing stage must be done only once, whatever the number of lassos to be generated. Easy computations show that the memory space requirement is $O(n \times |V|)$ integer numbers, where $|V|$ denotes the number of vertices in G . The number of arithmetic operations needed for the preprocessing stage is in the worst case in $O(nd|V|)$, where d stands for the maximum number of transitions from a state; and the generation stage is $O(nd)$ [13]. However,

the memory required is too large for very long lassos. In [18] we presented an improved version, named Dichopile, which avoids to have the whole table in memory, leading to a space requirement in $O(|V| \log n)$, at the price of a time requirement in $O(nd|V| \log n)$.

4 Application to LTL Model-Checking

This section shows how to use this generation algorithm (and its variants, see [18]) for LTL randomised model-checking. We note M and Φ the considered model and LTL formula. We propose a randomised method to check whether $M \models \Phi$. We call B_M a Büchi automaton corresponding to M : the transitions are labeled, the underlying graph is assumed to be reducible, and all the states are accepting (but this condition can be relaxed). We call $B_{\neg \Phi}$ the Büchi automaton corresponding to the negation of Φ . The problem is to check that $L(B) = \emptyset$ where $B = B_M \times B_{\neg \Phi}$.

It is possible to avoid the construction of the product $B = B_M \times B_{\neg \Phi}$. The idea is to exploit the fact that this product is the result of a total synchronisation between B_M and $B_{\neg \Phi}$: as explained in [4], the behaviours of B are exactly those behaviours of B_M accepted by $B_{\neg \Phi}$. It means that a lasso in B corresponds to a lasso in B_M (but not the reverse). Therefore, it is possible to draw lassos from B_M and, using $B_{\neg \Phi}$ as an observer, to reject those lassos that are not in B . It is well-known that such rejections preserve uniformity in the considered subset [16].

4.1 Drawing Lassos in B

There is a pre-processing phase that contains the one described in Section 3, namely:

(pre-DFS) the collection of the set B_E of back edges via a DFS in B_M , and computing n , the depth of the DFS + 1;

(pre-vector) the construction of the vector of the $|V|$ values $f_u(1)$, i.e. the numbers of lassos of length 1 starting for every vertex u ;

The two steps above are only needed once for each model. They are independent of the properties to be checked. The third step below is dependent on the property:

(pre-construction of the negation automaton) the construction of the Büchi automaton $B_{\neg \Phi}$.

Lassos are drawn from B_M using the Dichopile algorithm [18] and then observed with $B_{\neg \Phi}$ to check whether they are lassos of B . Moreover, it is also checked whether an acceptance state of $B_{\neg \Phi}$ is traversed during the cycle. The observation may yield three possible results:

- the lasso is not a lasso in $B_M \times B_{\neg \Phi}$;
- the lasso is an accepting lasso in $B_M \times B_{\neg \Phi}$;
- the lasso is a non-accepting lasso in $B_M \times B_{\neg \Phi}$.

Observation of lassos

The principle of the observation algorithm is: given a lasso of B_M , and the $B_{\neg \Phi}$ automaton, the algorithm explores $B_{\neg \Phi}$ guided by the lasso: since $B_{\neg \Phi}$ is generally non deterministic, the algorithm performs a traversal of a tree made of prefixes of the lasso.

When progressing in $B_{\neg \Phi}$ along paths of this tree, the algorithm notes whether the state where the cycle of the lasso starts and comes back has been traversed; when it has been done, it notes whether an accepting state of the automaton is met.

The algorithm terminates either when it reaches the end of the lasso or when it fails to reach it: it is blocked after having explored all the strict prefixes of the lasso present in $B_{\neg \Phi}$. The first case means that the lasso is also a lasso of $B_M \times B_{\neg \Phi}$; then if an accepting state of $B_{\neg \Phi}$ has been seen in the cycle of the lasso, it is an accepting lasso. Otherwise it is a non-accepting lasso. The second case means that the lasso is not a lasso in $B_M \times B_{\neg \Phi}$.

As soon as an accepting lasso is found, the drawing is stopped.

4.2 Complexities

Given a formula Φ and a model M , let $|\Phi|$ the length of formula Φ , $|V|$ the number of states of B_M and $|E|$ its number of transitions, the complexities of the pre-processing treatments are the following:

- (**pre-DFS**) this DFS is performed in B_M ; it is $O(|E|)$ in time and $O(|V|)$ in space in the worst case;
- (**pre-vector**) the construction of the vector of the $f_u(1)$ is $\Theta(|V|)$ in time and space;
- (**pre-construction of the negation automaton**) the construction of $B_{\neg \Phi}$ is $O(2^{|\Phi|} \cdot \log |\Phi|)$ in time and space in the worst case [24].

The drawing of one lasso of length n in B_M using the Dichopile algorithm is $O(n.d.|V|. \log n)$ in time and $O(|V|. \log n)$ in space; then its observation is a DFS of maximum depth n in a graph whose size can reach $O(2^{|\Phi|})$. Let $d_{B_{\neg \Phi}}$ the maximal out-degree of $B_{\neg \Phi}$, the worst case time complexity is $\min(d_{B_{\neg \Phi}}^n, 2^{|\Phi|})$, and the space complexity is $O(n)$.

The main motivation for randomised model-checking is gain in space. With this respect, using isotropic random walks as in [11] is quite satisfactory since a local knowledge of the model is sufficient. However, it may lead to bad coverage of the model. For instance, [11] reports the case of the Needham-Schroeder protocol where a counter-exemple has a very low probability to be covered by isotropic random walk. The solution presented here avoids this problem, since it ensures a uniform drawing of lassos, but it requires more memory: $\Theta(|V|)$ for the pre-processing and $O(|V|. \log n)$ for the generation.

We can also compare the complexity of our approach to the complexity of practical algorithm for the model checking of LTL. The NDFS algorithm [5] has a space complexity of $O(r \log |V|)$ for the main (randomly accessed) memory,

where r is the number of reachable states. This complexity is obtained thanks to the use of hash tables. Information accessed in a more structured way (e.g. sequentially) are stored on an external memory and retrieved using prediction and cache to lower the access cost.

In this case, uniform sampling of lassos, with a space complexity of $O(|V| \log n)$ is close to the $O(r \log |V|)$ of the classic NDFS algorithm, without being exhaustive. Nevertheless, a randomized search has no bias in the sequence of nodes it traverses (as is NDFS), and may lead faster to a counterexample in practice.

4.3 Probabilities

Let lassos_B the number of lassos in B , and lassos_{B_M} the number of lassos in B_M , i. e. $\text{lassos}_{B_M} = f_s(n)$, we have $\text{lassos}_B \leq f_s(n)$. The probability for a lasso to be rejected by the observation is $\text{lassos}_B / f_s(n)$, where the value of lassos_B is dependent on the considered LTL formula. Thus, the average time complexity for drawing N lassos in B is the complexity of the pre-processing, which is $O(|E|) + O(2^{|\Phi|})$ plus $N \times f_s(n) / \text{lassos}_B$ [16] times the complexity of drawing one lasso in B_M and observing it, which is $O(n \log n |V|) + \min(d_{B_{\neg\phi}}^n, 2^{|\Phi|})$.

Since the drawing in B_M is uniform, and $f_s(n)$ is the number of lassos the probability to draw any lasso in B_M is $1/f_s(n)$. Since there are less lassos in B , and rejection preserves uniformity [16], the probability to draw any lasso in B is $1/\text{lassos}_B$ which is greater or equal to $1/f_s(n)$. It is true for any lasso in B , the accepting or the non accepting ones. Thus there is no accepting lasso with too low probability as it was the case in the Needham-Schroeder example in [11].

Moreover, the probability ρ that $M \models \Phi$ after N drawings of non-accepting lassos is greater than:

$$1 - (1 - 1/f_s(n))^N$$

Increasing N may lead to high probabilities. Conversely, the choice of N may be determined by a target probability ρ :

$$N \geq \frac{\log(1 - \rho)}{\log(1 - 1/f_s(n))} \quad (1)$$

Remark: A natural improvement of the method is to use the fact that during the preliminary DFS some lassos of B_M are discovered, namely one by back edge. These lassos can be checked as above for early discovery of accepting lassos in $B_M \times B_{\neg\phi}$. However, it is difficult to state general results on the gain in time and space, since this gain is highly dependent on the topology of the graph.

5 Experimental Results

In this section, we report results about first experiments, which use the algorithm presented in Section 4 to verify if an LTL property holds on some models. We

first chose a model in which it is difficult to find a counter-example with isotropic random walks. The idea is to evaluate the cost-effectiveness of our algorithm: does it find a counter-example in a reasonable amount of time and memory? How many lassos have to be checked before we get a high probability that $M \models \Phi$? [2].

5.1 Implementation and Methodology

This algorithm has been implemented using the RUKIA library¹. This C++ library proposes several algorithms to generate uniformly at random paths in automata. In particular, the Dichopile algorithm that is mentioned in Section 3. We also use several tools that we mention here: The Boost Graph Library (BGL) for classical graph algorithms like the depth-first search algorithm; The GNU Multiple Precision (GMP) and Boost.Random libraries for generating random numbers; The LTL2BA software [9] to build a Büchi automaton from a LTL formula.

We did all our experiments on a dedicated server whose hardware is composed of an Intel Xeon 2.80GHz processor with 16GB memory. Each experiment was performed 5 times. The two extreme values are discarded and the three remaining values are averaged.

5.2 Description of the Model and the Formula

Figure 3 shows B_M , the Büchi automaton of the model. It has q states and every state, except s_q , has two transitions labeled by the action a : stay in the current state or move to the next state. The state s_q can only do the action $\neg a$.

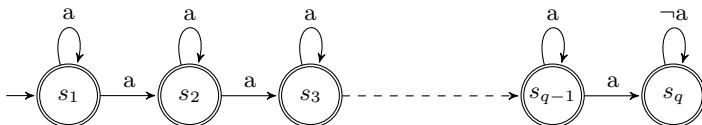


Fig. 3. The Büchi automaton, B_M , of the model. Its transitions are labeled by a or $\neg a$, to indicate if the action a occurs or not.

The property that we want to check on this model is: “an action a should occur infinitely often”. In LTL, this property can be expressed as

$$\phi = GFa,$$

where the operator G means “for all” and the operator F means “in the future”.

It is clear that $M \not\models \Phi$ because if s_q is reached, then no a will occur. Hence, there is a behavior in M where the action a will not occur infinitely often. However, it is difficult for an isotropic random walk to find the lasso that traverses s_q . In the next section, we measure the difficulty to find this lasso with our algorithm.

¹ <http://rukia.lri.fr>

5.3 LTL Model-Checking with Uniform Generation of Lassos

The first step of the algorithm is to build a Büchi automaton that represents the formula $\neg\phi$ (here, $\neg\phi = FG\neg a$). We used the tool LTL2BA and got the automaton in Figure 4.

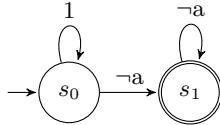


Fig. 4. The Büchi automaton, $B_{\neg\phi}$, of the formula $\neg\phi$. The label 1 means that both a and $\neg a$ are accepted.

Then, we can apply our algorithm to generate lassos in B_M until we find an accepting lasso in $B_M \cap B_{\neg\phi}$, by observing the lasso of B_M with the automaton $B_{\neg\phi}$. Table 2 shows the time needed to our algorithm to find the counter-example in two versions of the automaton in Figure 3, one with $q = 100$ states and the other with $q = 1000$ states. We also tried to find this counter-example with isotropic random walks. It did not work, even after the generation of 2 billions lassos. Thus, using uniform generation of lassos provides a better detection power of counter-examples.

Table 2. $M \not\models \Phi$: elapsed time, used memory and numbers of lassos generated in B_M by the algorithm of Section 4 to find the counter-example of Section 5.2

# states	Time (s)	Mem (MiB)	Nb lassos
100	0.38	49	70
1000	546	50	680

As we know the probability to find the counter-example in B with both isotropic random walks and uniform random generation (i.e., $1/2^q$ for an isotropic random walk and $1/q$ for a uniform random generation), we can compute the number of lassos required to achieve a target probability ρ . Table 3 describes those numbers for some target probabilities and for the two previous versions of the automaton in Figure 3.

Note: In general, the minimal probability to find a counter example is unknown. In the case of uniform random generation of lassos, we have a lower bound of this probability. Thus, the maximal number of lassos to be generated for a given probability can always be determined with Formula 1, which it is not possible with isotropic random walks.

Table 3. $M \models \Phi$: numbers N of lassos to be generated with isotropic random walks (resp. uniform random generation) in order to ensure a probability ρ that $M \models \Phi$. The symbol ∞ means a huge number, which cannot be computed with a calculator.

# states	ρ	N	
		isotropic	uniform
100	0.9	10^{30}	227
	0.99	∞	454
	0.999	∞	681
1000	0.9	∞	2300
	0.99	∞	4599
	0.9	∞	23022

6 Conclusion

We have presented a randomised approach to LTL model checking that ensures a uniform distribution on the lassos in the product $B = B_M \times B_{\neg \Phi}$. Thus, there is no accepting lasso with too low probability to be traversed, whatever the topology of the underlying graph.

As presented here, the proposed algorithm still needs an exhaustive traversal of the state graph during the pre-processing stage. This could seriously limit its applicability. A first improvement of the method would be to use on the fly techniques to avoid the storage in memory of the whole model during the DFS. A second improvement would be to avoid the complete storage of the vector, parts of it being computed during the generation stage, when needed. However, it will somewhat increase the time complexity of drawing. Another possibility would be approximate lasso counting, thus approximate uniformity, under the condition that the approximation error can be taken into account in the estimation of the satisfaction probability, which is an open issue.

First experiments, on examples known to be pathological, show that the method leads to a much better detection power of counter-examples, and that the drawing time is acceptable. In Section 5, we report a case with long counter-examples difficult to reach by isotropic random walks. A counter-example is discovered after a reasonable number of drawings, where isotropic exploration would require prohibitive numbers of them. The method needs to be validated on some more realistic example. We plan to embed it in an existing model-checker in order to check LTL properties on available case studies.

The method is applicable to models where the underlying graph is a reducible flow graph: we give a method for counting lassos and drawing them at random in this class of graphs, after recalling that it is a hard problem in general. Reducible data flow graphs correspond to well-structured programs and control-command systems (i.e., the steam boiler [2]). A perspective of improvement would be to alleviate the requirement of reducibility. It seems feasible: for instance, some data flow analysis algorithms have been generalised to communicating automata

in [15]. Similarly, we plan to study the properties and numbers of lassos in product of reducible automata, in order to consider multi-threaded programs.

References

1. Abed, N., Tripakis, S., Vincent, J.-M.: Resource-aware verification using randomized exploration of large state spaces. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 214–231. Springer, Heidelberg (2008)
2. Abrial, J.-R., Börger, E., Langmaack, H. (eds.): Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar). LNCS, vol. 1165. Springer, Heidelberg (1996)
3. Aldous, D.: An introduction to covering problems for random walks on graphs. *J. Theoret. Probab.* 4, 197–211 (1991)
4. Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P.: Systems and Software Verification. In: Model-Checking Techniques and Tools, Springer, Heidelberg (2001)
5. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. *Formal methods in system design* 1(2), 275–288 (1992)
6. Denise, A., Gaudel, M.-C., Gouraud, S.-D.: A generic method for statistical testing. In: 15th International Symposium on Software Reliability Engineering (ISSRE 2004), pp. 25–34. IEEE Computer Society, Los Alamitos (2004)
7. Denise, A., Gaudel, M.-C., Gouraud, S.-D., Lassaigne, R., Peyronnet, S.: Uniform random sampling of traces in very large models. In: 1st International ACM Workshop on Random Testing, pp. 10–19 (July 2006)
8. Dwyer, M.B., Elbaum, S.G., Person, S., Purandare, R.: Parallel randomized state-space search. In: 29th International Conference on Software Engineering (ICSE 2007), pp. 3–12 (2007)
9. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
10. Gaudel, M.-C., Denise, A., Gouraud, S.-D., Lassaigne, R., Oudinet, J., Peyronnet, S.: Coverage-biased random exploration of large models. In: 4th ETAPS Workshop on Model Based Testing. Electronic Notes in Theoretical Computer Science, vol. 220(1,10), pp. 3–14 (2008) (invited lecture)
11. Grosu, R., Smolka, S.A.: Monte Carlo model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 271–286. Springer, Heidelberg (2005)
12. Hecht, M.S., Ullman, J.D.: Characterizations of reducible flow graphs. *J. ACM* 21(3), 367–375 (1974)
13. Hickey, T., Cohen, J.: Uniform random generation of strings in a context-free language. *SIAM J. Comput.* 12(4), 645–655 (1983)
14. Johnson, D.B.: Finding all the elementary circuits of a directed graph. *SIAM J. Comput.* 4(1), 77–84 (1975)
15. Labb  , S., Gallois, J.-P.: Slicing communicating automata specifications: polynomial algorithms for model reduction. *Formal Asp. Comput.* 20(6), 563–595 (2008)
16. Devroye, L.: Non-Uniform Random Variate Generation. Springer, Heidelberg (1986)

17. Oudinet, J.: Uniform random walks in very large models. In: RT 2007: Proceedings of the 2nd International Workshop on Random Testing, pp. 26–29. ACM Press, New York (2007)
18. Oudinet, J., Denise, A., Gaudel, M.-C.: A new dichotomic algorithm for the uniform random generation of words in regular languages. In: Conference on random and exhaustive generation of combinatorial objects (GASCom), Montreal, Canada, p. 10 (September 2010)
19. Pelánek, R., Hanzl, T., Černá, I., Brim, L.: Enhancing random walk state space exploration. In: Proc. of Formal Methods for Industrial Critical Systems (FMICS 2005), Lisbon, Portugal, pp. 98–105. ACM Press, New York (2005)
20. Rungta, N., Mercer, E.G.: Generating counter-examples through randomized guided search. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 39–57. Springer, Heidelberg (2007)
21. Sivaraj, H., Gopalakrishnan, G.: Random walk based heuristic algorithms for distributed memory model checking. In: Proc. of Parallel and Distributed Model Checking (PDMC 2003). Electr. Notes Theor. Comput. Sci., vol. 89(1) (2003)
22. Tarjan, R.E.: Enumeration of the elementary circuits of a directed graph. SIAM J. Comput. 2(3), 211–216 (1973)
23. Tiernan, J.C.: An efficient search algorithm to find the elementary circuits of a graph. Commun. ACM 13(12), 722–726 (1970)
24. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1996)
25. Weinblatt, H.: A new search algorithm for finding the simple cycles of a finite directed graph. J. ACM 19(1), 43–56 (1972)
26. West, C.H.: Protocol validation in complex systems. In: SIGCOMM 1989: Symposium proceedings on Communications architectures & protocols, pp. 303–312. ACM, New York (1989)

Model Checking Büchi Pushdown Systems

Juncao Li¹, Fei Xie¹, Thomas Ball², and Vladimir Levin²

¹ Department of Computer Science, Portland State University
Portland, OR 97207, USA

{juncao,xie}@cs.pdx.edu

² Microsoft Corporation
Redmond, WA 98052, USA
{tball, vladlev}@microsoft.com

Abstract. We develop an approach to model checking Linear Temporal Logic (LTL) properties of Büchi Pushdown Systems (BPDS). Such BPDS models are suitable for Hardware/Software (HW/SW) co-verification. Since a BPDS represents the asynchronous transitions between hardware and software, some transition orders are unnecessary to be explored in verification. We design an algorithm to reduce BPDS transition rules, so that these transition orders will not be explored by model checkers. Our reduction algorithm is applied at compile time; therefore, it is also suitable to runtime techniques such as co-simulation. As a proof of concept, we have implemented our approach in our co-verification tool, CoVer. CoVer not only verifies LTL properties on the BPDS models represented by Boolean programs, but also accepts assumptions in LTL formulae. The evaluation demonstrates that our reduction algorithm can reduce the verification cost by 80% in time usage and 35% in memory usage on average.

1 Introduction

Hardware/Software (HW/SW) co-verification, verifying hardware and software together, is essential to establishing the correctness of complex computer systems. In previous work, we proposed a Büchi Pushdown System (BPDS) as a formal representation for co-verification [1]: a Büchi Automaton (BA) represents a hardware device model and a Labeled Pushdown System (LPDS) represents a model of the system software; the interactions between hardware and software take place through the synchronization of the BA and LPDS. This is different from a BPDS model used in software verification [2], where BA only monitors the state transitions of the Pushdown System (PDS) (see Related Work). We also designed an algorithm for checking safety properties of BPDS [1,3]. However, besides the verification of safety properties, the verification of liveness properties is also highly desirable. For example, a driver and its device should not hang on an I/O operation; a reset command from a driver should eventually reset the device.

We present an approach to LTL model checking of BPDS and design a reduction algorithm to reduce the verification cost. Given an LTL formula φ to be checked on a BPDS \mathcal{BP} , we constructed a BA \mathcal{B}_φ from $\neg\varphi$ to monitor the state transitions of \mathcal{BP} . The model checking process computes if \mathcal{B}_φ has an accepting run on \mathcal{BP} . Since a BPDS has two asynchronous components, i.e., a BA and an LPDS, we design our

model checking algorithm in such a way that the fairness between them are guaranteed. We also design an algorithm to reduce the BPDS transition rules based on the concept of static partial order reduction [4]. Our reduction algorithm is applied at compile time when constructing a BPDS model rather than during model checking; therefore, the algorithm is also suitable to runtime techniques such as co-simulation [5]. Different from other partial order reduction techniques [4,6], our approach can reduce many visible transitions without affecting the LTL_{-X} properties¹ to be verified, which is very effective in reducing the co-verification cost.

As a proof of concept, we have implemented our approach in our co-verification tool, CoVer. CoVer not only verifies LTL properties on the BPDS models represented by Boolean programs [7], but also accepts assumptions in LTL formulae. These assumptions are very helpful in practice to constrain the verification and rule out false positives. We have also designed an evaluation template to generate BPDS models with various complexities. The evaluation demonstrates that our reduction algorithm can reduce the verification cost by 80% in time usage and 35% in memory usage on average.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 introduces the background of this paper. Section 4 presents our LTL model checking algorithm for BPDS. Section 5 elaborates on our reduction algorithm. Section 6 presents the implementation details of CoVer and illustrates an example of BPDS represented by Boolean programs. Section 7 presents the evaluation results. Section 8 concludes and discusses future work.

2 Related Work

Bouajjani, et al. [8] presented a procedure to compute backward reachability (a.k.a., *pre**) of PDS and apply this procedure to linear/branching-time property verification. This approach was improved by Schwoon [2], which results in a tool, Moped, for checking LTL properties of PDS. An LTL formula is first negated and then represented as a BA, which is combined with the PDS to monitor its state transitions; therefore, the model checking problem is to compute if the BA has an accepting run. The goal of the previous research was to verify software only; however, our goal is co-verification.

Cook, et al. [9] presented an approach to termination checking of system code through proofs. The approach has two phases: first constructing the termination argument which is a set of ranking functions and then proving that one of the ranking functions decreases between the pre- and post-states of all finite transition sequences in the program. When checking the termination of a device driver, its hardware behavior is necessary to be modeled; otherwise, the verification may report a false positive or miss a real bug (see examples in Section 6).

Device Driver Tester (DDT) [5] is a symbolic simulation engine for testing closed-source binary device drivers against undesired behaviors, such as race conditions, memory errors, resource leaks, etc. Given driver's binary code, it is first reverse-engineered and then simulated with symbolic hardware, a shallow hardware model that mimics simple device behaviors such as interrupts. When simulating the interactions between device and driver, DDT employs a reduction method that allows interrupts only after

¹ LTL_{-X} is the subset of the logic LTL without the next time operator.

each kernel API call by the driver to operate the hardware device. While the reduction method of DDT was not formally justified, such kind of reduction can be formalized as the static partial reduction approach discussed in this paper.

Our previous work [3] of co-verification implemented an automatic reachability analysis algorithm for BPDS models specified using the C language. The concept of static partial order reduction is applied to reduce the complexity of the BPDS model only for reachability analysis. However, no algorithm was designed for either co-verification of liveness properties or its complexity reduction.

3 Background

3.1 Büchi Automaton (BA)

A BA \mathcal{B} [10] is a non-deterministic finite state automaton accepting infinite input strings. Formally, $\mathcal{B} = (\Sigma, Q, \delta, q_0, F)$, where Σ is the input alphabet, Q is the finite set of states, $\delta \subseteq (Q \times \Sigma \times Q)$ is the set of state transitions, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. \mathcal{B} accepts an infinite input string if and only if (iff) it has a run over the string that visits at least one of the final states infinitely often. A run of \mathcal{B} on an infinite string s is a sequence of states visited by \mathcal{B} when taking s as the input. We use $q \xrightarrow{\sigma} q'$ to denote a transition from state q to q' with the input symbol σ .

3.2 Labeled Pushdown System (LPDS)

An LPDS \mathcal{P} [1] is a tuple $(I, G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle)$, where I is the input alphabet, G is a finite set of global states, Γ is a finite stack alphabet, $\Delta \subseteq (G \times \Gamma) \times I \times (G \times \Gamma^*)$ is a finite set of transition rules, and $\langle g_0, \omega_0 \rangle$ is the initial configuration. An LPDS transition rule is written as $\langle g, \gamma \rangle \xleftarrow{\tau} \langle g', w \rangle \in \Delta$, where $\tau \in I$. A configuration of \mathcal{P} is a pair $\langle g, \omega \rangle \in G \times \Gamma^*$. The set of all configurations is denoted as $Conf(\mathcal{P})$. The head of a configuration $c = \langle g, \gamma v \rangle$ ($\gamma \in \Gamma, v \in \Gamma^*$) is $\langle g, \gamma \rangle$ and denoted as $head(c)$. Similarly the head of a rule $r = \langle g, \gamma \rangle \xleftarrow{\tau} \langle g', \omega \rangle$ is $\langle g, \gamma \rangle$ and denoted as $head(r)$. Given the same rule r , for every $v \in \Gamma^*$, the immediate successor relation is denoted as $\langle g, \gamma v \rangle \xrightarrow{r} \langle g', \omega v \rangle$, where we say this state transition follows the LPDS rule r . The reachability relation, \Rightarrow^* , is the reflexive and transitive closure of the immediate successor relation. A path of \mathcal{P} on an infinite input string, $\tau_0 \tau_1 \dots \tau_i \dots$, is written as $c_0 \xrightarrow{\tau_0} c_1 \xrightarrow{\tau_1} \dots c_i \xrightarrow{\tau_i} \dots$, where the path is also referred to as a trace of \mathcal{P} if $c_0 = \langle g_0, \omega_0 \rangle$ is the initial configuration.

3.3 Büchi Pushdown System (BPDS)

A BPDS \mathcal{BP} , as defined in [1], is the Cartesian product of a BA \mathcal{B} and an LPDS \mathcal{P} , where the input alphabet of \mathcal{B} is the power set of the set of propositions that may hold on a configuration of \mathcal{P} ; the input alphabet of \mathcal{P} is the power set of the set of propositions that may hold on a state of \mathcal{B} ; and two labeling functions are defined as follows:

- $L_{\mathcal{P}2\mathcal{B}} : (G \times \Gamma) \rightarrow \Sigma$, associates the head of an LPDS configuration with the set of propositions that hold on it. Given a configuration $c \in \text{Conf}(\mathcal{P})$, we write $L_{\mathcal{P}2\mathcal{B}}(c)$ instead of $L_{\mathcal{P}2\mathcal{B}}(\text{head}(c))$ for simplicity.
- $L_{\mathcal{B}2\mathcal{P}} : Q \rightarrow I$, associates a state of \mathcal{B} with the set of propositions that hold on it.

There are three definitions that help the presentation of BPDS:

Enabledness. A BA transition $t = q \xrightarrow{\sigma} q' \in \delta$ is enabled by an LPDS configuration c (resp. an LPDS rule $r = c \xleftarrow{\tau} c' \in \Delta$) iff $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(c)$; otherwise t is disabled by c (resp. r). The LPDS rule r is enabled by the BA state q (resp. the BA transition t) iff $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q)$; otherwise, r is disabled by q (resp. t).

Indistinguishability. Given a BA transition $t = q \xrightarrow{\sigma} q' \in \delta$, an LPDS rule $r = c \xleftarrow{\tau} c' \in \Delta$ is indistinguishable to t iff $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(c) \cap L_{\mathcal{P}2\mathcal{B}}(c')$, i.e., t is enabled by both c and c' . On the other hand, t is indistinguishable to r iff $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q) \cap L_{\mathcal{B}2\mathcal{P}}(q')$, i.e., r is enabled by both q and q' .

Independence. Given a BA transition t and an LPDS rule r , if they are indistinguishable to each other, t and r are independent; otherwise if either t or r is not indistinguishable to the other but they still enable each other, t and r are dependent. The independence relation is symmetric.

A BPDS $\mathcal{BP} = ((G \times Q), \Gamma, \Delta', \langle (g_0, q_0), \omega_0 \rangle, F')$ is constructed by taking the Cartesian product of \mathcal{B} and \mathcal{P} . A configuration of \mathcal{BP} is denoted as $\langle (g, q), \omega \rangle \in (G \times Q) \times \Gamma^*$. The set of all configurations is denoted as $\text{Conf}(\mathcal{BP})$. $\langle (g_0, q_0), \omega_0 \rangle$ is the initial configuration. For all $g \in G$ and $\gamma \in \Gamma$, $\langle (g, q), \gamma \rangle \in F'$ if $q \in F$. If we strictly follow the idea of Cartesian product, a BPDS rule in Δ' is constructed from a BA transition in δ and an LPDS rule in Δ ; therefore, both BA and LPDS have to transition simultaneously so that BPDS can make a transition. In order to model the asynchronous executions between BA and LPDS, we also need to introduce self-loops to BA and LPDS respectively. The set of BPDS rules, Δ' , is constructed as follows: given a BA transition $t = q \xrightarrow{\sigma} q' \in \delta$ and an LPDS rule $r = \langle g, \gamma \rangle \xleftarrow{\tau} \langle g', \omega \rangle \in \Delta$ that enable each other,

- if r and t are dependent, add $\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega \rangle$ to Δ' , i.e., \mathcal{B} and \mathcal{P} transition together.
- otherwise, add three rules to Δ' : (1) \mathcal{B} transitions and \mathcal{P} self-loops, i.e., $\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g, q'), \gamma \rangle$; (2) \mathcal{P} transitions and \mathcal{B} self-loops, i.e., $\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q), \omega \rangle$; and (3) \mathcal{B} and \mathcal{P} transition together, i.e., $\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega \rangle$.

The head of a configuration $c = \langle (g, q), \gamma v \rangle$ ($\gamma \in \Gamma, v \in \Gamma^*$) is $\langle (g, q), \gamma \rangle$ and denoted as $\text{head}(c)$. Similarly the head of a rule $r = \langle (g, q), \gamma \rangle \xleftarrow{\tau} \langle (g', q'), \omega \rangle$ is $\langle (g, q), \gamma \rangle$ and denoted as $\text{head}(r)$. Given the same rule r , for every $v \in \Gamma^*$, the immediate successor relation in BPDS is denoted as $\langle (g, q), \gamma v \rangle \Rightarrow_{\mathcal{BP}} \langle (g', q'), \omega v \rangle$, where we say this state transition follows the BPDS rule r . The reachability relation, $\Rightarrow_{\mathcal{BP}}^*$, is the reflexive and transitive closure of the immediate successor relation. A path of \mathcal{BP} is a sequence of BPDS configurations, $\pi = c_0 \Rightarrow_{\mathcal{BP}} c_1 \dots \Rightarrow_{\mathcal{BP}} c_i \Rightarrow_{\mathcal{BP}} \dots$, where π satisfies both the Büchi constraint and the BPDS loop constraint. Büchi constraint requires that if π is infinitely long, it should have infinite many occurrences of BPDS configurations from the set $\{ c \mid \text{head}(c) \in F' \}$. Given that

- the projection of π on \mathcal{B} , denoted as $\pi^{\mathcal{B}}$, is a sequence of state transitions of \mathcal{B} , and
- the projection of π on \mathcal{P} , denoted as $\pi^{\mathcal{P}}$, is a path of \mathcal{P} ,

BPDS loop constraint requires that if π is infinite, both $\pi^{\mathcal{B}}$ and $\pi^{\mathcal{P}}$ should also be infinite. Since self-loop transitions are introduced to \mathcal{B} and \mathcal{P} when constructing BPDS, we define BPDS loop constraint as a fairness constraint to guarantee that neither \mathcal{B} nor \mathcal{P} can self-loop infinitely on these self-loop transitions. The BPDS path π is also referred to as a trace of $\mathcal{B}\mathcal{P}$ if c_0 is the initial configuration.

4 Model Checking Algorithms for BPDS

4.1 Model Checking Problem

Our goal is to verify LTL properties on BPDS. Given a BPDS $\mathcal{B}\mathcal{P}$, an LTL formula φ , and a labeling function $L_{\varphi} : \text{Conf}(\mathcal{B}\mathcal{P}) \rightarrow 2^{At(\varphi)}$ that associates a BPDS configuration to a set of propositions that are true of it ($At(\varphi)$ is the set of atomic propositions in φ), there exists a BA $\mathcal{B}_{\varphi} = (2^{At(\varphi)}, Q_{\varphi}, \delta_{\varphi}, q_{\varphi 0}, F_{\varphi})$ that accepts the language $\mathcal{L}(\neg\varphi)$; therefore we can synthesize a transition system, $\mathcal{B}^2\mathcal{P}$, from $\mathcal{B}\mathcal{P}$ and \mathcal{B}_{φ} , where conceptually, \mathcal{B}_{φ} monitors the state transitions of $\mathcal{B}\mathcal{P}$.

We construct $\mathcal{B}^2\mathcal{P} = (G \times Q \times Q_{\varphi}, \Gamma, \Delta_{\mathcal{B}^2\mathcal{P}}, \langle (g_0, q_0, q_{\varphi 0}), \omega_0 \rangle, F_{\mathcal{B}^2\mathcal{P}})$, where $G \times Q \times Q_{\varphi}$ is the finite set of global states, Γ is the stack alphabet, $\Delta_{\mathcal{B}^2\mathcal{P}}$ is the finite set of transition rules, $\langle (g_0, q_0, q_{\varphi 0}), \omega_0 \rangle$ is the initial configuration, and $F_{\mathcal{B}^2\mathcal{P}} = F' \times F_{\varphi}$. The transition relation $\Delta_{\mathcal{B}^2\mathcal{P}}$ is constructed such that $(c, q_{\varphi}) \xrightarrow{\mathcal{B}^2\mathcal{P}} (c', q'_{\varphi}) \in \Delta_{\mathcal{B}^2\mathcal{P}}$ iff $c \xrightarrow{\mathcal{B}\mathcal{P}} c' \in \Delta'$, $q_{\varphi} \xrightarrow{\sigma} q'_{\varphi} \in \delta_{\varphi}$, and $\sigma \subseteq L_{\varphi}(c)$. The set of all configurations is denoted as $\text{Conf}(\mathcal{B}^2\mathcal{P}) \subseteq G \times Q \times Q_{\varphi} \times \Gamma^*$. For the purpose of simplicity, we also write $\mathcal{B}^2\mathcal{P} = (P, \Gamma, \Delta_{\mathcal{B}^2\mathcal{P}}, F_{\mathcal{B}^2\mathcal{P}})$, where $P = G \times Q \times Q_{\varphi}$. The head of a configuration $c = \langle p, \gamma v \rangle$ ($\gamma \in \Gamma, v \in \Gamma^*$) is $\langle p, \gamma \rangle$ and denoted as $\text{head}(c)$. Similarly the head of a rule $r = \langle p, \gamma \rangle \xrightarrow{\mathcal{B}^2\mathcal{P}} \langle p', \omega \rangle$ is $\langle p, \gamma \rangle$ and denoted as $\text{head}(r)$. The immediate successor relation and reachability relation are denoted respectively as $\Rightarrow_{\mathcal{B}^2\mathcal{P}}$ and $\Rightarrow_{\mathcal{B}^2\mathcal{P}}^*$. A path of $\mathcal{B}^2\mathcal{P}$ is written as $c_0 \Rightarrow_{\mathcal{B}^2\mathcal{P}} c_1 \Rightarrow_{\mathcal{B}^2\mathcal{P}} \dots$, where the path is also referred to as a trace of $\mathcal{B}^2\mathcal{P}$ if c_0 is the initial configuration.

Definition 1. An accepting run of $\mathcal{B}^2\mathcal{P}$ is an infinite trace π such that (1) π has infinitely many occurrences of configurations from the set $\{ c \mid \text{head}(c) \in F_{\mathcal{B}^2\mathcal{P}} \}$, i.e., the Büchi acceptance condition is satisfied; and (2) both $\pi^{\mathcal{B}}$ and $\pi^{\mathcal{P}}$ are infinite, i.e., the BPDS loop constraint is satisfied.

Definition 2. Given a BPDS $\mathcal{B}\mathcal{P}$ and an LTL formula φ , the model checking problem is to compute if the $\mathcal{B}^2\mathcal{P}$ model constructed from $\mathcal{B}\mathcal{P}$ and φ has an accepting run.

4.2 Model Checking Algorithm

We define a binary relation $\Rightarrow_{\mathcal{B}^2\mathcal{P}}^r$ between two configurations of $\mathcal{B}^2\mathcal{P}$ as: $c \Rightarrow_{\mathcal{B}^2\mathcal{P}}^r c'$, iff $\exists \langle p, \gamma \rangle \in F_{\mathcal{B}^2\mathcal{P}}$ such that $c \Rightarrow_{\mathcal{B}^2\mathcal{P}}^* \langle p, \gamma v \rangle \Rightarrow_{\mathcal{B}^2\mathcal{P}}^+ c'$, where $v \in \Gamma^*$. A head $\langle p, \gamma \rangle$ is repeating if $\exists v \in \Gamma^*$ such that $\langle p, \gamma \rangle \Rightarrow_{\mathcal{B}^2\mathcal{P}}^r \langle p, \gamma v \rangle$. The set of repeating heads is denoted as $\text{Rep}(\mathcal{B}^2\mathcal{P})$. We refer to the path that demonstrates a repeating head as a repeating path.

² $\pi^{\mathcal{B}}$ and $\pi^{\mathcal{P}}$ do not contain any self-loop transitions introduced when constructing the BPDS.

Proposition 1. Given the initial configuration c_0 , $\mathcal{B}^2\mathcal{P}$ has an accepting run iff (1) $\exists c_0 \Rightarrow_{\mathcal{B}^2\mathcal{P}}^* c'$ such that $\text{head}(c') \in \text{Rep}(\mathcal{B}^2\mathcal{P})$; and (2) a repeating path π_s of $\text{head}(c')$ satisfies the condition that $|\pi_s^\mathcal{B}| \neq 0$ and $|\pi_s^\mathcal{P}| \neq 0$. (see [11] for proof)

Our LTL model checking algorithm for BPDS has two phases. First, computing a special set of repeating heads, $R \subseteq \text{Rep}(\mathcal{B}^2\mathcal{P})$, where the repeating paths of the heads satisfy the BPDS loop constraint. Second, checking if there exists a path of $\mathcal{B}^2\mathcal{P}$ that leads from the initial configuration to a configuration c such that $\text{head}(c) \in R$.

In the first phase, we compute R . We construct a head reachability graph $\mathcal{G} = ((P \times \Gamma), E)$, where the set of nodes are the heads of $\mathcal{B}^2\mathcal{P}$, the set of edges $E \subseteq (P \times \Gamma) \times \{0, 1\}^3 \times (P \times \Gamma)$ denotes the reachability relation between the heads. Given a rule $r \in \Delta_{\mathcal{B}^2\mathcal{P}}$, we define three labeling functions: (1) $F_{\mathcal{B}^2\mathcal{P}}(r) = 1$ if $\text{head}(r) \in F_{\mathcal{B}^2\mathcal{P}}$ and $F_{\mathcal{B}^2\mathcal{P}}(r) = 0$ if otherwise; (2) $R_{\mathcal{B}}(r) = 1$ if r is constructed using a BA transition from δ and $R_{\mathcal{B}}(r) = 0$ if otherwise; and (3) $R_{\mathcal{P}}(r) = 1$ if r is constructed using an LPDS rule from Δ and $R_{\mathcal{P}}(r) = 0$ if otherwise. An edge $(\langle p, \gamma \rangle, (b_1, b_2, b_3), \langle p', \gamma' \rangle)$ belongs to E iff $\exists r = \langle p, \gamma \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p'', v_1 \gamma' v_2 \rangle$ and $\exists \pi = \langle p'', v_1 \rangle \Rightarrow_{\mathcal{B}^2\mathcal{P}}^* \langle p', \varepsilon \rangle$, where $p, p', p'' \in P$, $\gamma, \gamma' \in \Gamma$, $v_1, v_2 \in \Gamma^*$, ε denotes the empty string, and:

- $b_1 = 1$, iff $F_{\mathcal{B}^2\mathcal{P}}(r) = 1$ or $\langle p'', v_1 \rangle \Rightarrow_{\mathcal{B}^2\mathcal{P}}^* \langle p', \varepsilon \rangle$;
- $b_2 = 1$, iff $R_{\mathcal{B}}(r) = 1$ or $|\pi^\mathcal{B}| \neq 0$;
- $b_3 = 1$, iff $R_{\mathcal{P}}(r) = 1$ or $|\pi^\mathcal{P}| \neq 0$;

This definition is based on the idea of backward reachability computation. Given the head $\langle p', \varepsilon \rangle$ reachable from $\langle p'', v_1 \rangle$, if there exists a rule to indicate that $\langle p'', v_1 \gamma' \rangle$ is reachable from $\langle p, \gamma \rangle$, then we know that the head $\langle p', \gamma' \rangle$ (a.k.a., $\langle p', \varepsilon \gamma' \rangle$) is reachable from the head $\langle p, \gamma \rangle$. During such a computation process, we use the three labels defined above to record the information whether a path between the heads contains a final state in $F_{\mathcal{B}^2\mathcal{P}}$ and satisfies the BPDS loop constraint.

The set R can be computed by exploiting the fact that a head $\langle p, \gamma \rangle$ is repeating and the repeating path satisfies the BPDS loop constraint iff $\langle p, \gamma \rangle$ is part of a Strongly Connected Component (SCC) of \mathcal{G} and this SCC has internal edges labeled by $(1, *, *)$, $(*, 1, *)$, and $(*, *, 1)$, where $*$ represents 0 or 1. Algorithm REPHEADS takes $\mathcal{B}^2\mathcal{P}$ as the input and computes the set R . REPHEADS first utilizes the backward reachability analysis algorithm of [2], a.k.a., pre^* , to compute the edges E of \mathcal{G} . Given $\Delta_{\mathcal{B}^2\mathcal{P}}$, pre^* finds a set of rules $\text{trans} \subseteq \Delta_{\mathcal{B}^2\mathcal{P}}$ such that trans has rules all in the form of $\langle p, \gamma \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p', \varepsilon \rangle$, also written as (p, γ, p') for simplicity. With the three labels defined above, we can further write a rule in trans as $(p, [\gamma, b_1, b_2, b_3], p')$. Given such a rule, the algorithm between line 7 and 19 computes the reachability relation between heads. Specifically, when we see a rule $\langle p_1, \gamma_1 \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma \rangle$ at line 11 or line 13, we know $\langle p', \varepsilon \rangle$ is reachable from $\langle p_1, \gamma_1 \rangle$, so we add a new rule to trans ; when we see a rule $\langle p_1, \gamma_1 \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma \gamma_2 \rangle$ at line 15, we know $\langle p', \gamma_2 \rangle$ is reachable from $\langle p_1, \gamma_1 \rangle$, so we add a new rule to Δ_{label} , where a rule in Δ_{label} describes the reachability relation between two heads through more than one transitions; and rel stores the processed rules from trans . Meanwhile, we also use the labels to record the information whether a final

REPHEADS($\mathcal{B}^2\mathcal{P} = (P, \Gamma, \Delta_{\mathcal{B}^2\mathcal{P}}, F_{\mathcal{B}^2\mathcal{P}})$)

```

1:  $rel \leftarrow \emptyset, trans \leftarrow \emptyset, \Delta_{label} \leftarrow \emptyset$ 
2:
3:  $\{First, compute the head reachability graph of \mathcal{B}^2\mathcal{P} using pre^*\}$ 
4: for all  $r = \langle p, \gamma \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p', \varepsilon \rangle \in \Delta_{\mathcal{B}^2\mathcal{P}}$  do
5:    $\{Add the labeled rule r (written in a simplified form) to trans\}$ 
6:    $trans \leftarrow trans \cup \{(p, [\gamma, F_{\mathcal{B}^2\mathcal{P}}(r), R_{\mathcal{B}}(r), R_{\mathcal{P}}(r)], p')\}$ 
7: while  $trans \neq \emptyset$  do
8:   pop  $t = (p, [\gamma, b_1, b_2, b_3], p')$  from  $trans$ ;
9:   if  $t \notin rel$  then
10:     $rel \leftarrow rel \cup \{t\};$ 
11:    for all  $r = \langle p_1, \gamma_1 \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma \rangle \in \Delta_{\mathcal{B}^2\mathcal{P}}$  do
12:       $trans \leftarrow trans \cup \{(p_1, [\gamma_1, b_1 \vee F_{\mathcal{B}^2\mathcal{P}}(r), b_2 \vee R_{\mathcal{B}}(r), b_3 \vee R_{\mathcal{P}}(r)], p')\}$ 
13:      for all  $\langle p_1, \gamma_1 \rangle \stackrel{l}{\hookrightarrow}_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma \rangle \in \Delta_{label},$  where  $l = (b'_1, b'_2, b'_3)$  do
14:         $trans \leftarrow trans \cup \{(p_1, [\gamma_1, b_1 \vee b'_1, b_2 \vee b'_2, b_3 \vee b'_3], p')\}$ 
15:        for all  $r = \langle p_1, \gamma_1 \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma \rangle \in \Delta_{\mathcal{B}^2\mathcal{P}}$  do
16:           $\Delta_{label} \leftarrow \Delta_{label} \cup \{\langle p_1, \gamma_1 \rangle \stackrel{l}{\hookrightarrow}_{\mathcal{B}^2\mathcal{P}} \langle p', \gamma_2 \rangle\},$  where
              $l = (b_1 \vee F_{\mathcal{B}^2\mathcal{P}}(r), b_2 \vee R_{\mathcal{B}}(r), b_3 \vee R_{\mathcal{P}}(r))$ 
17:           $\{Match the new rule with the rules that have been processed\}$ 
18:          for all  $\langle p', [\gamma_2, b'_1, b'_2, b'_3], p'' \rangle \in rel$  do
19:             $trans \leftarrow trans \cup \{(p_1, [\gamma_1, b_1 \vee b'_1 \vee F_{\mathcal{B}^2\mathcal{P}}(r),
               b_2 \vee b'_2 \vee R_{\mathcal{B}}(r), b_3 \vee b'_3 \vee R_{\mathcal{P}}(r)], p'')\}$ 
20:  $R \leftarrow \emptyset, E \leftarrow \emptyset$ 
21:  $\{Direct reachability between two heads, i.e., indicated by a rule of \mathcal{B}^2\mathcal{P}\}$ 
22: for all  $r = \langle p, \gamma \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p', \gamma' v \rangle \in \Delta_{\mathcal{B}^2\mathcal{P}},$  where  $v \in \Gamma^*$  do
23:    $E \leftarrow E \cup \{(\langle p, \gamma \rangle, (F_{\mathcal{B}^2\mathcal{P}}(r), R_{\mathcal{B}}(r), R_{\mathcal{P}}(r)), \langle p', \gamma' \rangle)\}$ 
24:  $\{Indirect reachability between two heads, i.e., computed by pre^*\}$ 
25: for all  $\langle p, \gamma \rangle \stackrel{l}{\hookrightarrow}_{\mathcal{B}^2\mathcal{P}} \langle p', \gamma' \rangle \in \Delta_{label}$  do
26:    $E \leftarrow E \cup \{(\langle p, \gamma \rangle, l, \langle p', \gamma' \rangle)\}$ 
27:
28:  $\{Second, find R in \mathcal{G}\}$ 
29: Find strongly connected components,  $SCC,$  in  $\mathcal{G} = ((P \times \Gamma), E)$ 
30: for all  $C \in SCC$  do
31:   if  $C$  has edges labeled by  $(1, *, *), (*, 1, *),$  and  $(*, *, 1),$  where  $*$  represents 0 or 1 then
32:      $\{C contains repeating heads whose repeating paths satisfy the BPDS loop constraint\}$ 
33:      $R \leftarrow R \cup \{\text{the heads in } C\}$ 
34: return  $R$ 

```

state is found and the BPDS loop constraint is satisfied on the path between two heads. Second, the algorithm detects all the SCCs in \mathcal{G} and checks if one of the SCCs contains repeating heads (required by the label $(1, *, *)$) and satisfies the BPDS constraint (required by the two labels $(*, 1, *)$ and $(*, *, 1)$).

Theorem 1. Algorithm REPHEADS takes $O(|P|^2|\Delta_{\mathcal{B}^2\mathcal{P}}|)$ time and $O(|P||\Delta_{\mathcal{B}^2\mathcal{P}}|)$ space. (see [11] for proof)

In the second phase, after R is computed, we compute $post^*(\{c_0\}) \cap R,$ i.e., given the initial configuration $c_0, \exists c_0 \Rightarrow^* c'$ such that $head(c') \in R.$ The forward reachability algorithms, a.k.a., $post^*,$ for PDS-equivalent models have been well studied. We use the

forward reachability algorithm [2] with a complexity of $O((|P| + |\Delta_{\mathcal{B}^2\mathcal{P}}|)^3)$. Therefore, the LTL model checking of BPDS has the complexity of $O((|P| + |\Delta_{\mathcal{B}^2\mathcal{P}}|)^3)$.

5 Reduction

We present how to utilize the concept of static partial order reduction in the LTL_X checking of BPDS. As illustrated in Figure 1, our reduction is based on the observation that when \mathcal{B} and \mathcal{P} transition asynchronously, one can run while the other one self-loops. Figure 1(a) is a complete state transition graph. There are three types of transition edges: (1) a horizontal edge represents a transition when \mathcal{B} transitions and \mathcal{P} self-loops; (2) a vertical edge represents a transition when \mathcal{P} transitions and \mathcal{B} self-loops; and (3) a diagonal edge represents a transition when \mathcal{B} and \mathcal{P} transition together. If the graph satisfies certain requirements (see below), we can reduce many state transitions while preserving the LTL_X property to be checked as illustrated in

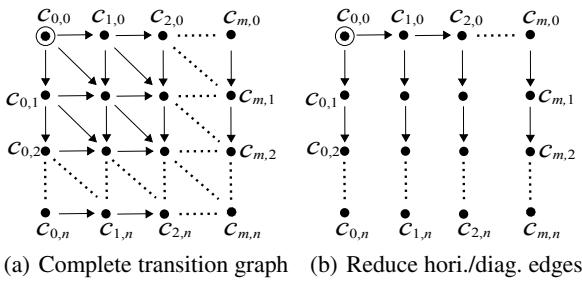


Fig. 1. An example of reducing state transition edges

Figure 1(b). The reduced BPDS is denoted as \mathcal{BP}_r , where only the BPDS rules are reduced.

Definition 3. Given a BPDS rule r , $\text{VisProp}(r)$ denotes the set of propositional variables whose value is affected by r . If $\text{VisProp}(r) = \emptyset$, r is said to be invisible.

Definition 4. Given a labeling function L , two infinite paths $\pi_1 = s_0 \rightarrow s_1 \rightarrow \dots$ and $\pi_2 = q_0 \rightarrow q_1 \rightarrow \dots$ are stuttering equivalent, denoted as $\pi_1 \sim_{st} \pi_2$, if there are two infinite sequences of positive integers $0 = i_0 < i_1 < i_2 < \dots$ and $0 = j_0 < j_1 < j_2 < \dots$ such that for every $k \geq 0$, $L(s_{i_k}) = L(s_{i_k+1}) = \dots = L(s_{i_{k+1}-1}) = L(q_{j_k}) = L(q_{j_k+1}) = \dots = L(q_{j_{k+1}-1})$ [6].

It is already known that any LTL_X property is invariant under stuttering [6]; therefore, given a trace π of \mathcal{BP} , we want to guarantee that there always exists a trace of \mathcal{BP}_r stuttering equivalent to π .

Given $t = q \xrightarrow{\sigma} q' \in \delta$ and $a \in 2^{At(\varphi)}$, for every $r = \langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g, q'), \gamma' \rangle \in \Delta'$, if $\text{VisProp}(r) = a \neq \emptyset$, t is said to be *horizontally visible*. Intuitively, horizontal visibility describes the situation when propositional variables are evaluated only based on the states of BA. This can help reduce many visible BPDS rules without affecting the LTL_X properties to be verified, since such a horizontal transition can be shifted on a BPDS trace to construct another stuttering equivalence trace. Given a BA transition t and an LPDS rule r , Algorithm REDUCIBLERULES decides whether the corresponding diagonal/horizontal BPDS rules are reducible candidates. We should assume that t and

REDUCIBLERULES($t \in \delta, r \in \Delta$)

Require: t and r are independent.

```

1:  $ReduceDiag \leftarrow \text{FALSE}$ ,  $ReduceHori \leftarrow \text{FALSE}$ 
2: Let  $t = q \rightarrow q'$ ,  $r = \langle g, \gamma \rangle \xleftarrow{\tau} \langle g', \omega \rangle$ 
3:    $r_1 = \langle (g, q), \gamma \rangle \hookrightarrow_{BP} \langle (g, q'), \gamma \rangle$  {Horizontal BPDS rules, see Figure 1(a)}
4:    $r_2 = \langle (g, q), \gamma \rangle \hookrightarrow_{BP} \langle (g', q), \omega \rangle$  {Vertical BPDS rules, see Figure 1(a)}
5:    $r_3 = \langle (g, q), \gamma \rangle \hookrightarrow_{BP} \langle (g', q'), \omega \rangle$  {Diagonal BPDS rules, see Figure 1(a)}
6: if  $VisProp(r_1) = \emptyset$  and  $VisProp(r_2) = \emptyset$  and  $VisProp(r_3) = \emptyset$  then
7:   {If  $r_1$ ,  $r_2$ , and  $r_3$  are all invisible}
8:    $ReduceDiag \leftarrow \text{TRUE}$ ,  $ReduceHori \leftarrow \text{TRUE}$ 
9: else
10:  if  $VisProp(r_1) = VisProp(r_3)$  or  $VisProp(r_2) = VisProp(r_3)$  or
       $VisProp(r_1) = \emptyset$  or  $VisProp(r_2) = \emptyset$  then
11:     $ReduceDiag \leftarrow \text{TRUE}$ 
12:  if  $r_1$  is invisible or  $t$  is horizontally visible then
13:     $ReduceHori \leftarrow \text{TRUE}$ 
14: return ( $ReduceDiag$ ,  $ReduceHori$ )

```

r are independent; otherwise, since \mathcal{B} and \mathcal{P} must transition together when t and r are dependent, no BPDS rule can be reduced. In this algorithm, at line 8, if there is no visible BPDS rules, both the horizontal rule r_1 and the diagonal rule r_3 are reducible candidates; at line 11, r_3 is a reducible candidate if it is replaceable by horizontal and vertical rules; at line 13, r_1 is a reducible candidate if it is either invisible or constructed from a BA transition (i.e., t) that is horizontally visible.

Definition 5. We define three sets of heads, $SensitiveSet$, $VisibleSet$, and $LoopSet$ on $Conf(\mathcal{P})$, as follows:

- $SensitiveSet = \{ head(\langle g_0, \omega_0 \rangle) \} \cup \{ head(c') \mid \exists r = c \xrightarrow{\tau} c' \in \Delta, \exists t \in \delta, r \text{ and } t \text{ are dependent} \},$ where $\langle g_0, \omega_0 \rangle$ is the initial configuration of \mathcal{P} ;
- $VisibleSet = \{ head(\langle g', \omega \rangle) \mid \exists r = \langle (g, q), \gamma \rangle \hookrightarrow_{BP} \langle (g', q'), \omega \rangle \in \Delta' \text{ visible to } \varphi; \text{ and } r \text{ is irreducible according to Algorithm REDUCIBLERULES} \};$
- $LoopSet = \{ h \mid \text{for every SCC } C \text{ in } \mathcal{G}_{\mathcal{P}}, \text{ pick a head } h \text{ from } C \},$ where $\mathcal{G}_{\mathcal{P}}$ is the head reachability graph of \mathcal{P} and there is no preference on how h is selected.

$SensitiveSet$ is introduced to preserve the reachability [3]; the concept of $VisibleSet$ is similar to that of $SensitiveSet$, i.e., preserving the reachability of BPDS paths right after a visible transition that is not reduced according REDUCIBLERULES; $LoopSet$, similar to the concept of cycle closing condition [4], is introduced to satisfy the BPDS loop constraint when a loop of \mathcal{P} is involved in the accepting run.

Algorithm BPDSRULESVIASPOR applies the reduction following the idea illustrated in Figure 1(b), where the horizontal/diagonal edges are reduced. At line 6, since the LPDS rule r and the BA transition t are dependent, \mathcal{B} and \mathcal{P} must transition together; at line 9, we construct a vertical rule to represent the asynchronous situation when \mathcal{P} transitions and \mathcal{B} self-loops. Since BPDSRULESVIASPOR follows the reduction idea of Figure 1(b), all vertical BPDS rules are preserved; at line 10, we invoke REDUCIBLERULES, to decide if the horizontal/diagonal BPDS rules are reducible candidates; at line 13, we construct a diagonal BPDS rule if necessary; at line 16, we

BPDSRULESVIASPOR($\delta \times \Delta$)

```

1:  $\Delta_{sync} \leftarrow \emptyset$ ,  $\Delta_{vert} \leftarrow \emptyset$ ,  $\Delta_{hori} \leftarrow \emptyset$ ,  $\Delta_{diag} \leftarrow \emptyset$ 
2: for all  $r = \langle g, \gamma \rangle \xrightarrow{\tau} \langle g', \omega \rangle \in \Delta$  do
3:   for all  $t = q \xrightarrow{\sigma} q' \in \delta$  and  $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(\langle g, \gamma \rangle)$  and  $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q)$  do
4:     if  $r$  and  $t$  are dependent then
5:       { $\mathcal{B}$  and  $\mathcal{P}$  must transition together}
6:        $\Delta_{sync} \leftarrow \Delta_{sync} \cup \{ \langle (g, q), \gamma \rangle \xrightarrow{\mathcal{B}\mathcal{P}} \langle (g', q'), \omega \rangle \}$ 
7:     else
8:       { $\mathcal{P}$  transitions and  $\mathcal{B}$  self-loops}
9:        $\Delta_{vert} \leftarrow \Delta_{vert} \cup \{ \langle (g, q), \gamma \rangle \xrightarrow{\mathcal{B}\mathcal{P}} \langle (g', q), \omega \rangle \}$ 
10:      ( $ReduceDiag, ReduceHori$ )  $\leftarrow$  REDUCIBLERULES( $t, r$ )
11:      if  $ReduceDiag = \text{FALSE}$  then
12:        { $\mathcal{B}$  and  $\mathcal{P}$  transition together}
13:         $\Delta_{diag} \leftarrow \Delta_{diag} \cup \{ \langle (g, q), \gamma \rangle \xrightarrow{\mathcal{B}\mathcal{P}} \langle (g', q'), \omega \rangle \}$ 
14:      if  $ReduceHori = \text{FALSE}$  or
15:         $\langle g, \gamma \rangle \in SensitiveSet \cup VisibleSet \cup LoopSet$  then
16:          { $\mathcal{B}$  transitions and  $\mathcal{P}$  self-loops}
17:           $\Delta_{hori} \leftarrow \Delta_{hori} \cup \{ \langle (g, q), \gamma \rangle \xrightarrow{\mathcal{B}\mathcal{P}} \langle (g, q'), \gamma \rangle \}$ 
18: return  $\Delta'_r$ 

```

construct a horizontal BPDS rule if necessary. Note that even if REDUCIBLERULES returns TRUE for $ReduceHori$, we still have to preserve this horizontal BPDS rule if $head(r)$ belongs to $SensitiveSet$, $VisibleSet$, or $LoopSet$.

Theorem 2. Algorithm BPDSRULESVIASPOR preserves all LTL_X properties to be verified on $\mathcal{B}\mathcal{P}$. (see [11] for proof.)

Complexity analysis. In BPDSRULESVIASPOR, let n_{sync} be the number of BPDS rules that are generated from dependent BA transitions and LPDS rules (at line 6), n_v be the number of BPDS rules related to visible transition rules (i.e., when REDUCIBLERULES returns FALSE for $ReduceDiag$ or $ReduceHori$), n_{svl} be the number of BPDS rules associated to $SensitiveSet$, $VisibleSet$, and $LoopSet$ (at line 16 when $ReduceHori$ is TRUE). We have $|\Delta_{hori} \cup \Delta_{diag}| = n_v + n_{svl}$ and $|\Delta_{sync}| = n_{sync}$. As illustrated in Figure 1, asynchronous transitions can be organized as triples where each one includes a vertical transition, a horizontal transition, and a diagonal transition, so we have $|\Delta_{vert}| = \frac{|\delta \times \Delta| - n_{sync}}{3}$. The number of rules generated by BPDSRULESVIASPOR is $|\Delta'_r| = n_{sync} + \frac{|\delta \times \Delta| - n_{sync}}{3} + n_v + n_{svl} = \frac{2}{3}n_{sync} + \frac{|\delta \times \Delta|}{3} + n_v + n_{svl}$. The number of transition rules reduced is $|\Delta'| - |\Delta'_r| = \frac{2}{3}|\delta \times \Delta| - n_v - \frac{2}{3}n_{sync} - n_{svl}$. Therefore, our reduction is effective when the following criteria have small sizes: (1) BPDS rules visible to φ ; (2) dependent transitions of \mathcal{B} and \mathcal{P} ; and (3) loops in \mathcal{P} .

6 Implementation

As a proof of concept, we have realized the LTL checking algorithm for BPDS as well as the static partial order reduction algorithm in our co-verification tool, CoVer. The implementation is based on the Moped model checker [2]. We specify the LPDS \mathcal{P} using

Boolean programs and the BA \mathcal{B} using Boolean programs with the semantic extension of relative atomicity [3], i.e., hardware transitions are modeled as atomic to software. In this section, we first present an example of a BPDS model specified in Boolean programs. Second, we illustrate how we specify LTL properties on such a BPDS model. Third, we elaborate on how CoVer generates a reduced BPDS model for the verification of an LTL $_X$ formula.

6.1 Specification of the BA \mathcal{B} and LPDS \mathcal{P}

We specify \mathcal{B} and \mathcal{P} using an approach similar to that described in [3], where the state transitions of \mathcal{B} are described by atomic functions. Figure 2 demonstrates such an example. The states of \mathcal{B} are represented by global variables. All the functions that are labeled

```

void main() begin
  decl v0,v1,v2 := 1,1,1;
  reset();
  // wait for the reset to complete
  v1,v0 := status();
  while(v1|v0) do v1,v0 := status(); od
  // wait for the counter to increase
  v2,v1,v0 := rd_reg();
  while(!v2) do v2,v1,v0 := rd_reg(); od
  // if the return value is valid
  if (v1|v0) then
    error: skip;
  fi
  exit: return;
end

// represent hardware registers
decl c0, c1, c2, r, s;
__atomic void inc_reg()
begin
  if (!c0) then c0 := 1;
  elseif (!c1) then c1,c0 := 1,0;
  elseif (!c2) then
    c2,c1,c0 := 1,0,0; fi
end

__atomic void reset()
begin reset_cmd: r := 1; end

__atomic bool<3> rd_reg()
begin return c2,c1,c0; end

__atomic bool<2> status()
begin return s,r; end

// hardware instrumentation function
void HWInstr() begin
  while(*) do HWModel(); od
end

// asynchronous hardware model
__atomic void HWModel() begin
  if (r) then
    reset_act: c2,c1,c0,r,s := 0,0,0,0,1;
    elseif(s) then inc_reg(); fi
end

```

Fig. 2. An example of \mathcal{B} and \mathcal{P} both specified in Boolean programs

by the keyword `__atomic` describe the state transitions of \mathcal{B} . Such kind of functions are also referred to as transaction functions. The function `main` models the behavior of \mathcal{P} , where `main` has three steps: (1) resets the state of \mathcal{B} by invoking the function `reset`; (2) waits for the reset to complete; (3) waits for the counter of \mathcal{B} to increase above 4, i.e., $v2==1$. When a transaction function, such as `reset` or `rd_reg`, is invoked from \mathcal{P} , it represents a dependent (a.k.a., synchronous) transition between \mathcal{B} and \mathcal{P} . On the other hand, the transaction function `HWModel` represents independent (a.k.a., asynchronous) transitions of \mathcal{B} with respect to \mathcal{P} . In this example, since the dependent transitions of \mathcal{B} and \mathcal{P} are already specified as direct function calls, the rest of the Cartesian product is to instrument \mathcal{P} with the independent transitions of \mathcal{B} , i.e., add function call to `HWInstr` after each statement in `main`.

6.2 Specification of LTL Properties

Without loss of generality, we specify LTL properties on the statement labels. For example, we write an LTL formula, $F \text{ exit}$, which asserts that the function `main` always terminates. This property is asserted on a very common scenario: when software waits for hardware to respond, the waiting thread should not hang. Verification of this property requires relatively accurate hardware models. As illustrated in Figure 2, the transaction function `HWModel` describes a hardware model that responds to software reset immediately; thus, the first while-loop in `main` will not loop for ever. Since the hardware

will start to increment its register after reset, the second while-loop will also terminate. Therefore, $F \text{exit}$ holds. Note that the non-deterministic while-loop in `HWInstr` will repeatedly call `HWModel1`, which is guaranteed by the BPDS loop constraint and the fairness between hardware state transitions (i.e., transitions specified by `HWModel1` should not be starved by self-loop transitions introduced when constructing a BPDS).

There may exist a hardware design that cannot guarantee immediate responses to software reset commands. Therefore, delays should be represented in the hardware model. Figure 3 illustrates a transaction function `HWModelSlow` which describes a hardware design that cannot guarantee immediate responses to reset commands.

```
atomic void HWModelSlow() begin
    if (r) then
        if (*) then reset_act: c2,c1,c0,r,s := 0,0,0,0,1; fi
        elsif(s) then inc_reg(); fi
    end
```

Fig. 3. Hardware does not respond to reset immediately

The property $F \text{exit}$ fails on the BPDS model that uses `HWModelSlow` for hardware, since the hardware can delay the reset operation infinitely. In practice, design engineers may want to assume that: hardware can delay the reset operation; therefore, software should wait for reset completion; however hardware should not delay the reset operation for ever. CoVer accepts such assumptions as LTL for-

mulæ. Under the assumption $G (\text{reset_cmd} \rightarrow (F \text{reset_act}))$, $F \text{exit}$ will hold on the BPDS model. Such kind of assumptions are also considered as the Büchi constraint specified on the hardware model.

As another example, we write an LTL formula, $G \text{!error}$, asserting that the labeled statement in `main` is not reachable. Verification of $G \text{!error}$ fails on the BPDS model in Figure 2. Since hardware is asynchronous with software when incrementing the register, it is impossible for software to control how fast the register is incremented. Therefore, when software breaks from the second while-loop, the hardware register may have already been incremented to 5, i.e., $(\text{v2} == 1) \&& (\text{v1} == 0) \&& (\text{v0} == 1)$.

6.3 Reduction during the Cartesian Product

In order to make the Cartesian product of \mathcal{B} and \mathcal{P} , we need to add function call to `HWInstr` after every software statement. As discussed in Section 5, certain BPDS transitions are unnecessary to be generated for such a product, i.e., it is unnecessary to call `HWInstr` after every software statement to verify an LTL_x property. We define the concrete counterparts corresponding to the concepts defined on $Conf(\mathcal{P})$:

Software synchronization points [3]. Corresponding to *SensitiveSet*, software synchronization points are defined as a set of program locations where the program statements right before these locations may be dependent with some of the hardware state transitions. In general, there are three types of software synchronization points: (1) the point where the program is initialized; (2) those points right after software reads/writes hardware interface registers; and (3) those points where hardware interrupts may affect the verification results. We may understand the third type in such a way that the effect of interrupts (by executing interrupt service routines) may influence certain program statements, e.g., the statements that access global variables.

Software visible points. Corresponding to $VisibleSet$, we define software visible points as a set of program locations right after the program statements whose labels are used in the LTL property. For example, in Figure 2 the program location right after the statement *error* can be a software visible point. However, the location right after the statement *reset_act* cannot be a software visible point, since this statement is in a transaction function for \mathcal{B} .

Software loop points. Corresponding to $LoopSet$, we define software loop points as a set of program locations involved in program loops. The precise detection of those loops needs to explore the program’s state graph, which is inefficient. Therefore, we try to identify a super set $LoopSet' \supseteq LoopSet$ using heuristics. A program location is included into the super set if it is at (1) the point right before the first statement of a while loop; (2) the point right before a backward goto statement; or (3) the entry of a recursive function, which can be detected by analyzing the call graph between functions.

As for implementation, CoVer first automatically detects the software synchronization points, visible points, and loop points in the Boolean program of \mathcal{P} and then inserts the function calls to `HWInstr` only at those detected points. Note that some transitions described by `HWModel` (called via `HWInstr`) may be visible when a statement

```

decl c0,c1,c2,r,s; // hardware registers
decl g; // software global variable
void main() begin
    decl v0,v1,v2 := 1,1,1;
    reset();
    v1,v0 := status();
    while(!v1|v0) do v1,v0 := status(); od
    // call the first level
    level<1>();
    v2,v1,v0 := rd_reg();
    while(!v2) do v2,v1,v0 := rd_reg(); od
    if (v1|v0) then error: skip; fi
    exit: return;
end

```

```

void level<i>()
begin
    decl v0,v1,v2,v3,v4,v5;
    v2,v1,v0 := rd_reg();
    v5,v4,v3 := rd_reg();
    v2,v1,v0 :=
        gcd<i>(v5,v4,v3,
                v2,v1,v0);
    if(*) then reset(); fi
    if(g) then
        g := (v3 != v0);
        <stmt>;
    fi
end

```

Fig. 4. The BPDS template $BPDS<N>$ for evaluation

label in `HWModel` is used in the LTL formula, e.g., $F !reset_act$. However, such BA transitions are horizontally visible, since *reset_act* is not affected by any transition of \mathcal{P} . This is why function calls to `HWInstr` can be reduced without affecting the LTL_X properties even if `HWModel` describes visible transitions. Compared to the trivial approach that inserts `HWInstr` after every software statement, our reduction can significantly reduce the complexity

of the verification model, since the number of the instrumentation points are usually very small in common applications.

7 Evaluation

We have designed a synthetic BPDS template $BPDS<N>$ for $N > 0$ to evaluate our algorithms. As illustrated in Figure 4, this template is similar to the BPDS in Figure 2. The major difference is between the models of \mathcal{P} . $BPDS<N>$ has two function templates `level<N>` and `gcd<N>` for \mathcal{P} , where each of the function templates has N instances. For $0 < i \leq N$, `level<i>` calls `gcd<i>` which is the i^{th} instance of `gcd<N>` that computes the greatest common divisor (implementation of `gcd<N>` is omitted).

Table 1. LTL checking of BPDS<N>

LTL Property	N (sec/MB)					
	500		1000		2000	
	No Reduction	Reduction	No Reduction	Reduction	No Reduction	Reduction
$F \text{ exit}$	177.9/49.1	55.6/27.8	606.8/98.1	100.9/55.6	1951.5/196.3	231.5/111.2
$G(\text{reset_cmd} \rightarrow (F \text{ reset_act}))$	100.8/51.1	19.2/31.6	439.0/102.1	37.2/63.2	1742.1/204.3	115.0/126.5
$F \text{ level_} N$	165.3/49.1	52.9/27.8	524.1/98.1	99.8/55.6	1934.1/196.3	230.7/111.2
$G \text{ !level_} N$	94.8/43.4	10.7/25.0	404.0/86.2	22.3/49.9	1728.9/172.5	84.5/99.9
$G \text{ !error}$	96.6/42.4	10.1/24.8	402.6/84.8	21.2/49.2	1719.9/169.8	81.5/98.5

For $0 < j < N$, the instance of $\langle \text{stmt} \rangle$ in the body of the function $\text{level}_{<j>}$ is replaced by a call to $\text{level}_{<j+1>}$. The instance of $\langle \text{stmt} \rangle$ in the body of $\text{level}_{<N>}$ is replaced by skip . The design of BPDS<N> mimics the common scenarios in co-verification: since hardware and software are mostly asynchronous, there are many software statements independent with hardware transitions.

Our evaluation runs on a Lenovo ThinkPad notebook with Dual Core 2.66GHz CPU and 4GB memory. Table 1 presents the statistics of verifying five LTL formulae on the BPDS models generated from BPDS<N>, where some of the LTL formulae are discussed in Section 6. The statistics suggests that our reduction algorithm can reduce the verification cost by 80% in time usage and 35% in memory usage on average.

Table 2 presents the statistics for the verification of BPDS models generated from BPDS_Slow<N>, a template that differs from BPDS<N> only in the hardware model. BPDS_Slow<N> uses the hardware model illustrated in Figure 3. As discussed in Section 6, the verification of the property $\mathcal{A}1$ or $\mathcal{A}2$ will fail on the BPDS models generated from BPDS_Slow<N>, since the hardware cannot guarantee an immediate response to the software reset command. However, by assuming $\mathcal{A}2$, the verification of $\mathcal{A}1$ should pass. Obviously, the verification of this property, denoted as φ (including both $\mathcal{A}1$ and $\mathcal{A}2$), costs more time and memory compared to other properties, because φ is more complex than other properties. Nevertheless, we can infer from the two tables that our reduction algorithm is very effective in reducing the verification cost. For example, without the reduction, verification of the property φ gets a spaceout failure for $N = 2000$, i.e., CoVer fails to allocate more memory from the Operating System.

Table 2. LTL checking of BPDS_Slow<N> using the hardware model of Figure 3

LTL Property	N (sec/MB)					
	500		1000		2000	
	No Reduction	Reduction	No Reduction	Reduction	No Reduction	Reduction
$\mathcal{A}1: F \text{ exit}$	186.5/49.1	38.1/27.8	576.4/98.1	98.5/55.6	1913.5/196.3	207.1/111.2
$\mathcal{A}2: G(\text{reset_cmd} \rightarrow (F \text{ reset_act}))$	143.1/61.0	28.3/35.5	587.1/122.0	64.3/71.0	1778.7/203.5	164.1/142.0
$\mathcal{A}1$ using $\mathcal{A}2$ as the assumption	1264.0/223.4	255.8/109.5	3750.3/446.7	565.6/218.9	N/A/spaceout	1260.8/437.7
$F \text{ level_} N$	181.9/49.1	42.2/27.8	588.6/98.1	90.8/55.6	1908.4/196.3	198.6/111.2
$G \text{ !level_} N$	96.7/43.4	12.1/25.0	414.6/86.2	26.9/49.9	1679.7/172.5	91.5/99.9
$G \text{ !error}$	95.0/42.5	11.5/24.8	414.2/84.8	25.3/49.2	1672.6/169.8	88.9/98.5

8 Conclusion and Future Work

We have developed an approach to LTL model checking of BPDS and designed a reduction algorithm to reduce the verification cost. As a proof of concept, we have implemented our approach in our co-verification tool, CoVer. CoVer not only verifies LTL properties on the BPDS models represented by Boolean programs, but also accepts assumptions in LTL formulae. The evaluation demonstrates that our reduction algorithm is very effective in reducing the verification cost.

Although illustrated using Boolean programs, our approach can also be applied with other programming languages such as C. In other words, the BA and LPDS can be described using the C language, and the Cartesian product can be made through instrumenting the software LPDS model with the hardware BA model (as used in [3]). However, one challenge to this approach is to support the efficient abstraction/refinement, since most loops need to be fully unrolled in liveness property checking. There are two options for future work: (1) implement an aggressive abstraction/refinement algorithm for loop computation in tools such as SLAM [7] (may be insufficient when a ranking function is required); or (2) utilize termination checking tools such as Terminator [9] which analyzes loops by checking termination arguments (i.e., ranking functions).

Acknowledgement. This research received financial support from National Science Foundation of the United States (Grant #: 0916968).

References

1. Li, J., Xie, F., Ball, T., Levin, V., McGarvey, C.: An automata-theoretic approach to hardware/software co-verification. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 248–262. Springer, Heidelberg (2010)
2. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, Technische Universität München, Institut für Informatik (2002)
3. Li, J., Xie, F., Ball, T., Levin, V.: Efficient Reachability Analysis of Büchi Pushdown Systems for Hardware/Software Co-verification. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 339–353. Springer, Heidelberg (2010)
4. Kurshan, R.P., Levin, V., Minea, M., Peled, D., Yenigün, H.: Static partial order reduction. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, p. 345. Springer, Heidelberg (1998)
5. Kuznetsov, V., Chipounov, V., Candea, G.: Testing closed-source binary device drivers with DDT. In: USENIX Annual Technical Conference (2010)
6. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press, Cambridge (1999)
7. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrussek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: EuroSys (2006)
8. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
9. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI (2006)
10. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press, Princeton (1994)
11. Li, J.: An Automata-Theoretic Approach to Hardware/Software Co-verification. PhD thesis, Portland State University (2010)

Modeling with Plausibility Checking: Inspecting Favorable and Critical Signs for Consistency between Control Flow and Functional Behavior

Claudia Ermel¹, Jürgen Gall¹, Leen Lambers², and Gabriele Taentzer³

¹ Technische Universität Berlin, Germany

`claudia.ermel@tu-berlin.de, jgall@cs.tu-berlin.de`

² Hasso-Plattner-Institut für Softwaresystemtechnik GmbH, Potsdam, Germany

`leen.lambers@hpi.uni-potsdam.de`

³ Philipps-Universität Marburg, Germany

`taentzer@informatik.uni-marburg.de`

Abstract. UML activity diagrams are a wide-spread modelling technique to capture behavioral aspects of system models. Usually, pre- and post-conditions of activities are described in natural language and are not formally integrated with the static domain model. Hence, early consistency validation of activity models is difficult due to their semi-formal nature. In this paper, we use integrated behavior models that integrate activity diagrams with object rules defining sets of actions in simple activities. We formalize integrated behavior models using typed, attributed graph transformation. It provides a basis for plausibility checking by static conflict and causality detection between specific object rules, taking into account their occurrence within the control flow. This analysis leads to favorable as well as critical signs for consistency of the integrated behavior model. Our approach is supported by ACTIGRA, an ECLIPSE plug-in for editing, simulating and analyzing integrated behavior models. It visualizes favorable and critical signs for consistency in a convenient way and uses the well-known graph transformation tool AGG for rule application as well as static conflict and causality detection. We validate our approach by modeling a conference scheduling system.

Keywords: graph transformation, activity model, plausibility, conflict and causality detection, object rule, AGG.

1 Introduction

In model-driven software engineering, models are key artifacts which serve as basis for automatic code generation. Moreover, they can be used for analyzing the system behavior prior to implementing the system. In particular, it is interesting to know whether integrated parts of a model are consistent. For behavioral models, this means to find out whether the modeled system actions are executable in general or under certain conditions only. For example, an action in a model run might prevent one of the next actions to occur because the preconditions

of this next action are not satisfied any more. This situation is usually called a *conflict*. Correspondingly, it is interesting to know which actions do depend on other actions, i.e. an action may be performed only if another action has occurred before. We call such situations *causalities*. The aim of this paper is to come up with a plausibility checking approach regarding the consistency of the control flow and the functional behavior given by actions bundled in object rules. Object rules define a pre-condition (which object pattern should be present) and a post-condition (what are the local changes). Intuitively, consistency means that for a given initial state there is at least one model run that can be completed successfully.

We combine activity models defining the control flow and object rules in an *integrated behavior model*, where an object rule is assigned to each simple activity in the activity model. Given a system state typed over a given class model, the behavior of an integrated behavior model can be executed by applying the specified actions in the pre-defined order. The new plausibility check allows us to analyze an integrated behavior model for *favorable* and *critical* signs concerning consistency. *Favorable* signs are e.g. situations where object rules are triggered by other object rules that precede them in the control flow. On the other hand, *critical* signs are e.g. situations where an object rule causes a conflict with a second object rule that should be applied after the first one along the control flow, or where an object rule depends causally on the effects of a second object rule which is scheduled by the control flow to be applied after the first one. An early feedback to the modeler indicating this kind of information in a natural way in the behavioral model is desirable to better understand the model.

In [10], sufficient consistency criteria for the executability of integrated behavior models have been developed. However, especially for an infinite set of potential runs (in case of loops), this technique may lead to difficulties. Moreover, it is based on sufficient criteria leading to false negatives. In this paper, we follow a different approach, focusing on *plausibility* reasoning on integrated behavior models and convenient visualization of the static analysis results. This approach is complementary to [10], since we opt for back-annotating light-weight static analysis results allowing for plausibility reasoning, also in case of lacking consistency analysis results from [10]¹.

This light-weight technique seems to be very appropriate to allow for early plausibility reasoning during development steps of integrated behavior models. We visualize the results of our plausibility checks in an integrated development environment called ACTIGRA². Potential inconsistencies and reasons for consistency are directly visualized within integrated behavior models, e.g. as colored arcs between activity nodes and by detailed conflict and causality views.

Structure of the paper: Section 2 presents our running example. In Section 3, we introduce our approach to integrated behavior modeling and review the underlying formal concepts for static analysis based on graph transformation as far as

¹ In [4], we explain in more detail how plausibility reasoning is related to the sufficient criteria in [10].

² <http://tfs.cs.tu-berlin.de/actigra>

needed. Different forms of plausibility checking are presented in Section 4, where we validate our approach checking a model of a conference scheduling system. A section on related approaches (Section 5) and conclusions including directions for future work (Section 6) close the paper.

2 Case Study: A Conference Scheduling System

This case study³ models planning tasks for conferences. Its class model is shown in Figure 1 (a). A *Conference* contains *Persons*, *Presentations*, *Sessions* and *Slots*. A *Person* gives one or more *Presentations* and may chair arbitrary many *Sessions*. Note that a session chair may give one or more presentations in the session he or she chairs. A *Presentation* is in at most one *Session* and *scheduled* in at most one *Slot*. Slots are linked as a list by *next* arcs and *used* by *Sessions*.

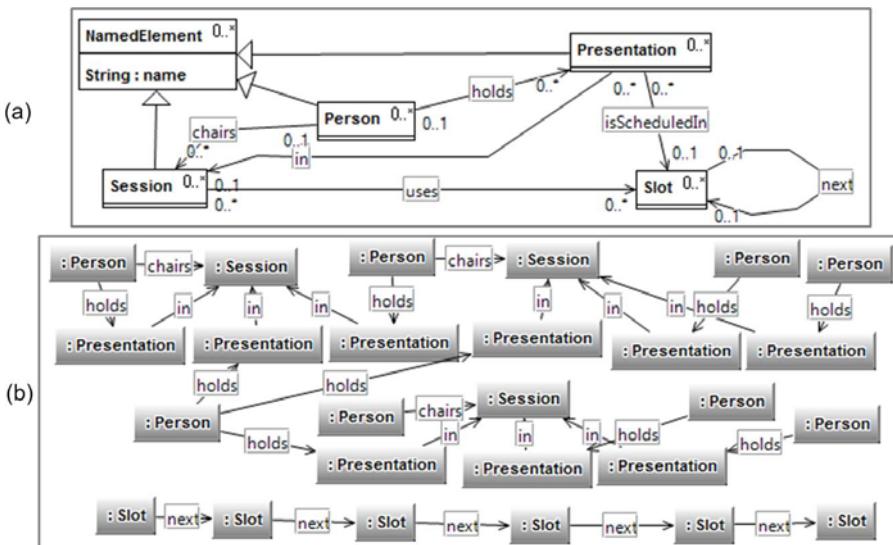


Fig. 1. Class and instance model for the *Conference Scheduling System*

Figure 1 (b) shows a sample object model of an initial session plan before presentations are scheduled into time slots⁴. This object model conforms to the class model. The obvious task is to find a valid assignment for situations like the one in Figure 1 (b) assigning the presentations to available time slots such that the following conditions are satisfied: (1) there are no simultaneous presentations given by the same presenter, (2) no presenter is chairing another session running simultaneously, (3) nobody chairs two sessions simultaneously, and (4)

³ Taken from the tool contest on *Graph-Based Tools 2008* [17].

⁴ Due to space limitations, we do not show name attributes here.

the presentations in one session are given not in parallel but in consecutive time slots. Moreover, it should be possible to generate arbitrary conference plans like the one in Figure 1 (b). This is useful to test the assignment procedure.

3 Integrating Activity Models with Object Rules

Our approach to behavior modeling integrates activity models with object rules, i. e. the application order of object rules is controlled by activity models. An object rule defines pre- and post-conditions of activities by sets of actions to be performed on object models. An object rule describes the behavior of a simple activity and is defined over a given class model. The reader is supposed to be familiar with object-oriented modelling using e.g. the UML [16]. Therefore, we present our approach to integrated behavior modeling from the perspective of its graph transformation-based semantics. In the following, we formalize class models by type graphs and object rules by graph transformation rules to be able to use the graph transformation theory [2] for plausibility checking.

3.1 Graphs and Graph Transformation

Graphs are often used as abstract representation of diagrams. When formalizing object-oriented modeling, graphs occur at two levels: the type level (defined based on class models) and the instance level (given by all valid object models). This idea is described by the concept of *typed graphs*, where a fixed *type graph* TG serves as an abstract representation of the class model. Types can be structured by an inheritance relation, as shown e.g. in the type graph for our *Conference Scheduling* model in Figure 1. Instance graphs of a type graph have a structure-preserving mapping to the type graph. The sample session plan in Figure 1 is an instance graph of the *Conference Scheduling* type graph.

Graph transformation is the rule-based modification of graphs. Rules are expressed by two graphs (L, R), where L is the left-hand side of the rule and R is the right-hand side. Rule graphs may contain variables for attributes. The left-hand side L represents the pre-conditions of the rule, while the right-hand side R describes the post-conditions. $L \cap R$ (the graph part that is not changed) and the union $L \cup R$ should form a graph again, i.e., they must be compatible with source, target and type settings, in order to apply the rule. Graph $L \setminus (L \cap R)$ defines the part that is to be deleted, and graph $R \setminus (L \cap R)$ defines the part to be created. Furthermore, the application of a graph rule may be restricted by so-called *negative application conditions* (NACs) which prohibit the existence of certain graph patterns in the current instance graph. Note that we indicate graph elements common to L and R or common to L and a NAC by equal numbers.

Figure 2 shows graph rule *initial-schedule* modeling the scheduling of the first presentation of some session to a slot. The numerous conditions for this scheduling step stated in Section 2 are modelled by 8 NACs. The NAC shown in Figure 2 means that the rule must not be applied if the presenter holds already another presentation in the same slot⁵.

⁵ For the complete case study with all rules and NACs see [1].

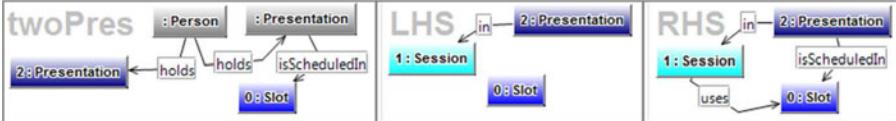


Fig. 2. Graph rule *initial-schedule*

A *direct graph transformation* $G \xrightarrow{r,m} H$ between two instance graphs G and H is defined by first finding a match m of the left-hand side L of rule r in the current instance graph G such that m is structure-preserving and type-compatible and satisfies the NACs (i.e. the forbidden graph patterns are not found in G). We use injective matches only. Attribute variables used in graph object $o \in L$ are bound to concrete attribute values of graph object $m(o)$ in G . The resulting graph H is constructed by (1) deleting all graph items from G that are in L but not also in R ; (2) adding all those new graph items that are in R but not also in L ; (3) setting attribute values of preserved and created elements.

A reason for non-determinism of graph transformation systems is the potential existence of several matches for one rule. If two rules are applicable to the same instance graph, they might be applicable in any order with the same result (parallel independence). If this is not the case, then we say that the corresponding rules are in *conflict*, since one rule *may disable* the other rule. If two rules are applicable one after the other to the same graph, it might be possible to switch their application order without changing the result (sequential independence). Conversely, it might be the case that one rule *may trigger* the application of another rule or *may be irreversible* after the application of another rule. In this case, this sequence of two rules is said to be causally dependent. See [13] for a formal description of conflict and causality characterizations⁶.

The static analysis of potential conflicts and causalities between rules is supported in AGG⁷, a tool for specifying, executing and analysing graph transformation systems. This analysis is based on critical pair analysis (CPA) [2,8] and critical sequence analysis (CSA) [13], respectively. Intuitively, each critical pair or sequence describes which rule elements need to overlap in order to cause a specific conflict or causality when applying the corresponding rules.

3.2 Integrated Behavior Models

As in [11], we define *well-structured activity models* as consisting of a start activity s , an activity block B , and an end activity e such that there is a transition between s and B and another one between B and e . An *activity block* can be a simple activity, a sequence of blocks, a fork-join structure, decision-merge structure, and loop. In addition, we allow complex activities which stand for nested well-structured activity models. In this hierarchy, we forbid nesting cycles. Activity blocks are connected by transitions (directed arcs). Decisions have

⁶ The different types of conflicts and causalities are reviewed also in [4].

⁷ AGG: <http://tfs.cs.tu-berlin.de/agg>

an explicit *if*-guard and implicit *else*-guard which equals the negated *if*-guard, and loops have a *loop*-guard with corresponding implicit *else*-guard.

In our formalization, an *integrated behavior model* is a well-structured activity model *A* together with a type graph such that each *simple activity* *a* occurring in *A* is equipped with a *typed graph transformation rule* *r_a* and each *if* or *loop* guard is either *user-defined* or equipped with a typed *guard pattern*. We have *simple* and *application-checking* guard patterns: a simple guard pattern is a graph that has to be found; an application-checking guard pattern is allowed for a transition entering a loop or decision followed by a simple activity in the loop-body or if-branch, respectively, and checks the applicability of this activity; it is formalized by a graph constraint [7] and visualized by the symbol $[*]$. User-defined guards are evaluated by the user at run time to true or false. An *initial state* for an integrated behavior model is given by a typed instance graph.

Example 1. Let us assume the system state shown in Figure 1 as initial state of our integrated behavior model. The activity diagram *ScheduleControl* is shown in the left part of Figure 3 (please disregard the colors for now). Its first step performs the initial scheduling of sessions and presentations into time slots by applying rule *initial-schedule* (see Figure 2) as long as possible.

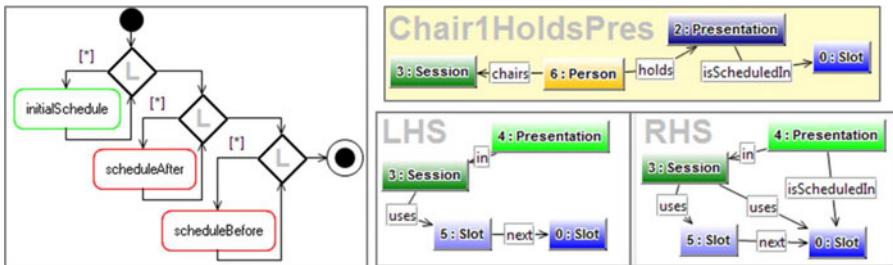


Fig. 3. Activity model *ScheduleControl* and rule *scheduleAfter*

As second step, two loops are executed taking care of grouping the remaining presentations of a session into consecutive time slots, i.e. a presentation is scheduled in a free time slot either directly before or after a slot where there is already a scheduled presentation of the same session. Rule *scheduleAfter* is shown in the right part of Figure 3. Rule *scheduleBefore* looks quite similar, only the direction of the next edge between the two slots is reversed. Both rules basically have the same NACs as rule *initialSchedule* ensuring the required conditions for the schedule (see [1]). The NAC shown here ensures that the session chair does not hold a presentation in the time slot intended for the current scheduling.

As in [11] we define a control flow relation on integrated behavior models.⁸ Intuitively, two activities or guards (*a*, *b*) are control flow-related whenever *b*

⁸ In contrast to [11], we include guards into the control flow relation.

is performed or checked after a . Moreover, we define an against-control flow relation which contains all pairs of activities or guards that are reverse to the control flow relation.

The *control flow relation* CFR_A of an activity model A contains all pairs (x, y) where x and y are activities or guards such that (1)-(4) holds: (1) $(x, y) \in CFR_A$ if there is a transition from activity x to activity y . (2) $(x, y) \in CFR_A$ if activity x has an outgoing transition with guard y . (3) $(x, y) \in CFR_A$ if activity y has an incoming transition with guard x . (4) If $(x, y) \in CFR_A$ and $(y, z) \in CFR_A$, then also $(x, z) \in CFR_A$. The *against-control flow relation* $ACFR_A$ of an activity model A contains all pairs (x, y) such that (y, x) is in CFR_A .

3.3 Simulation of Integrated Behavior Models

The *semantics* $Sem(A)$ of an integrated behavior model A consisting of a start activity s , an activity block B , and an end activity e is the *set of sequences* S_B , where each sequence consists of *rules alternated with graph constraints* (stemming from guard patterns), generated by the main activity block B (for a formal definition of the semantics see [11]).⁹ For a block being a simple activity a inscribed by rule r_a , $S_B = \{r_a\}$. For a sequence block $B = X \rightarrow Y$, we construct $S_B = S_X \text{ seq } S_Y$, i.e. the set of sequences being concatenations of a sequence in S_X and a sequence in S_Y . For decision blocks we construct the union of sequences of both branches (preceded by the if guard pattern and the negated guard pattern, respectively, in case that the if guard is not user-defined); for loop blocks we construct sequences containing the body of the loop i times ($0 \leq i \leq n$) (where each body sequence is preceded by the loop guard pattern and the repetition of body sequences is concluded with the negated guard pattern in case that the loop guard is not user-defined). In contrast to [11], we restrict fork-join-blocks to one simple activity in each branch and build a parallel rule from all branch rules [13,2].¹⁰ We plan to omit this restriction however, when integrating object flow [11] into our approach, since then it would be possible to build unique concurrent rules for each fork-join-branch. For B being a complex activity inscribed by the name of the integrated behavior model X , $S_B = Sem(X)$.

Given $s \in Sem(A)$ a sequence of rules alternated with graph constraints and a start graph S , representing an initial state for A . We then say that each graph transformation sequence starting with S , applying each rule to the current instance graph and evaluating each graph constraint to true for the current instance graph in the order of occurrence in s , represents a *complete simulation run* of A . An integrated behavior model A is *consistent* with respect to a start graph S , representing an initial state for A , if there is a sequence $s \in Sem(A)$ leading to a complete simulation run. In particular, if A contains user-defined guards, usually more than one complete simulation run should exist.

⁹ Note that $Sem(A)$ does not depend on the initial state of A . Moreover, we have a slightly more general semantics compared to [11], since we do not only have rules in the sequences of S_B , but also graph constraints.

¹⁰ This fork-join semantics is slightly more severe than in [11], which allows all interleavings of rules from different branches no matter if they lead to the same result.

In ACTIGRA we can execute simulation runs on selected activity models. Chosen activities are highlighted and the completion of simulation runs is indicated. User-defined guards are evaluated interactively. If a simulation run cannot be completed, an error message tells the user which activity could not be executed.

4 Plausibility Checks for Integrated Behavior Models

We now consider how to check plausibility regarding consistency of the control flow and the functional behavior given by actions bundled in object rules. Thereby, we proceed as follows: We characterize *desired properties* for an integrated behavior model and its initial state to be consistent. We determine the *favorable* as well as *critical signs*¹¹ for these properties to hold, show, how the checks are supported by ACTIGRA and illustrate by our case study which conclusions can be drawn by the modeler to validate our approach.

For the plausibility checks we wish to detect potential conflicts and causalities [4] between rules and guards occurring in the sequences of $Sem(A)$. Since in A simple activities, fork/joins as well as simple guard patterns correspond to rules¹² we just call them rules for simplicity reasons. Thereby, we disregard rules stemming from simple activities belonging to some fork/join block, since they do not occur as such in $Sem(A)$. Instead, the corresponding parallel rule for the fork/join is analyzed. As an exception to this convention, the plausibility check in Section 4.5 inspects consistency of fork/joins and analyzes also the enclosed simple activities.

4.1 Inspecting Initialization

If for some sequence in $Sem(A)$ the first rule is applicable, then the corresponding sequence can lead to a complete simulation run. Otherwise, the corresponding sequence leads to an incomplete run. Given an integrated behavior model A with initial state S , the first *plausibility check* computes automatically for which sequences in $Sem(A)$, the first rule is applicable to S . The modeler then may inspect the simulation run(s) that should complete for correct initialization

¹¹ In most cases, these favorable and critical signs merely describe *potential* reasons for the property to be fulfilled or not, respectively. For example, some critical pair describes which kind of rule overlap may be responsible for a critical conflict. By inspecting this overlap, the modeler may realize that the potential critical conflict may actually occur and adapt the model to avoid it. On the other hand, he may realize that it does not occur since the overlap corresponds to an invalid system state, intermediate rules deactivate the conflict, etc.

¹² For each *simple guard pattern* we can derive a *guard rule* (without side-effects) for the guarded branch and a negated guard rule for the alternative branch (as described in [11]). Application-checking guard patterns are evaluated for simulation but disregarded by the plausibility checks, since they are not independent guards but check for the application of succeeding *rules* only.

(*desired property*). We identify the *favorable signs* as the set of possible initializations: $FaI_A = \{r | r \text{ is first rule of sequence in } Sem(A) \text{ and } r \text{ is applicable to } S\}$. We identify the *critical signs* as the set of impossible initializations:

$CrI_A = \{r | r \text{ is first rule of a sequence in } Sem(A) \text{ and } r \text{ is not applicable to } S\}$.

ACTIGRA visualizes the result of this plausibility check by highlighting the elements of FaI_A in *green*. Rules belonging to CrI_A are highlighted in *red*¹³.

Example 2. Let us assume the system state in Figure 1 (b) as initial state. Figure 3 shows the initialization check result for activity model *ScheduleControl*. We have $FaI_{ScheduleControl} = \{initialSchedule\}$ and $CrI_{ScheduleControl} = \{scheduleAfter, scheduleBefore\}$. Thus, complete simulation runs on our initial state never start with *scheduleAfter* or *scheduleBefore*, but always with *initialSchedule*.

4.2 Inspecting Trigger Causalities along Control Flow Direction

If rule a may trigger rule b and b is performed after a , then it may be advantageous for the completion of a corresponding simulation run. If for some rule b no rule a is performed before b that may trigger b , this may lead to an incomplete simulation run and the modeler may decide to add some triggering rule or adapt the post-condition of some previous rule in order to create a trigger for b . Alternatively, the initial state could be adapted such that b is applicable to the start graph. Given an integrated behavior model A with initial state S , this *plausibility check* computes automatically for each rule a in A , which predecessor rules may trigger a . The modeler may inspect each rule a for enough predecessor rules to trigger a then (*desired property*). We identify the *favorable signs* as the set of potential trigger causalities for some rule a along control flow: $FaTrAl_A(a) = \{(b, a) | (b, a) \in CFR_A \text{ such that } b \text{ may trigger } a\}$. We say that $FaTrAl_A = \{FaTrAl_A(a) | a \text{ is a rule in } A\}$ is the *set of potential trigger causalities in A along control flow*. We identify the *critical signs* as the set of non-triggered rules along control flow that are not applicable to the initial state: $CrNonTrAl_A = \{a | a \text{ is rule in } A \text{ such that } FaTrAl_A(a) = \emptyset \text{ and } a \text{ is not applicable to } S\}$.

ACTIGRA visualizes the result of this plausibility check by displaying *dashed green arrows* from b to a selected rule a for each pair of rules (b, a) in $FaTrAl_A(a)$. If no rule is selected, then all pairs in $FaTrAl_A$ are displayed by dashed green arrows. Clicking on such an arrow from b to a opens a detail view, showing the reason(s) why b may trigger a as discovered by CSA. Conversely, ACTIGRA highlights each rule belonging to $CrNonTrAl_A$ in *red*.

Example 3. Consider activity model *GenConfPlans* in (Figure 4) for generating conference plans, assuming an empty initial state. The set of potential trigger causalities along control flow for *createSession* is given by $FaTrAl_{GenConfPlans}(createSession) = \{(createPerson + createPaper, createSession), (createPerson, createSession)\}$. Here, we learn that we need at least one execution of a loop

¹³ Concerning fork/join blocks in FaI_A or CrI_A , ACTIGRA colors the fork bar.

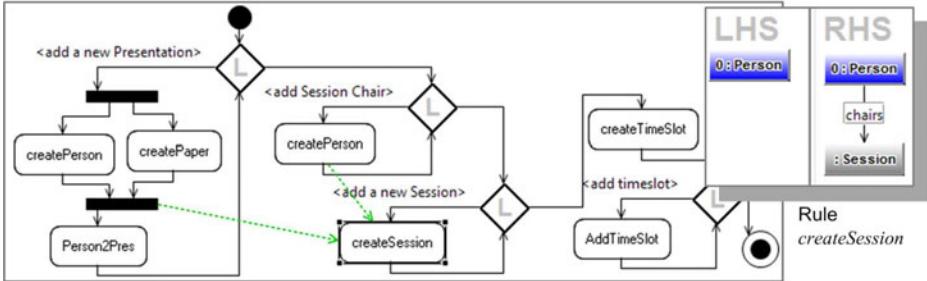


Fig. 4. Potential trigger causalities along control flow in activity model *GenConfPlans*

containing rule *createPerson* (a rule with an empty left-hand side) to ensure a complete simulation run containing *createSession*.

4.3 Inspecting Conflicts along Control Flow Direction

If rule *a* may disable rule *b*, and *b* is performed after *a*, then this may lead to an incomplete simulation run. On the other hand, if for some rule *a* no rule *b* performed before *a* exists that may disable rule *a*, then the application of *a* is not impeded. Given an integrated behavior model *A* with initial state *S*, this *plausibility check* computes automatically for each rule *a* in *A*, which successor rules *b* in *A* may be disabled by *a*. The modeler then may inspect each rule *a* in *A* for the absence of rules performed before *a* disabling rule *a* (*desired property*). We identify the *critical signs* as the set of potential conflicts along control flow caused by rule *a*: $CrDisAl_A(a) = \{(a, b) | a, b \text{ are rules in } A, (a, b) \in CFR_A \text{ and } a \text{ may disable } b\}$. We say that $CrDisAl_A = \{CrDisAl_A(a) | a \text{ is a rule in } A\}$ is the set of potential conflicts along control flow in *A*. We identify the *favorable signs* as the set of non-disabled rules along control flow: $FaNonDisAl_A = \{a | a \text{ in } A \text{ and } \nexists(b, a) \in CrDisAl_A\}$.

ACTiGRA visualizes the result of this plausibility check by displaying faint red arrows from *a* to *b* for each pair of rules (a, b) in $CrDisAl_A$. If rule *a* is selected, a bold red arrow from *a* to *b* for each pair of rules (a, b) in $CrDisAl_A(a)$ is shown. Clicking on such an arrow opens a detail view, showing the reason(s) why *a* may disable *b* as discovered by CPA. Each rule *a* in *A* belonging to $FaNonDisAl_A$ is highlighted in green.

Example 4. Consider activity model *SchedulingControl* in Figure 5 (a). Here, the set of potential conflicts along control flow caused by rule *initialSchedule* is given by $CrDisAl_{SchedulingControl}(initialSchedule) = \{(initialSchedule, initialSchedule), (initialSchedule, scheduleAfter), (initialSchedule, scheduleBefore)\}$ ¹⁴. This gives the modeler a hint that in fact a scheduling might not terminate successfully in the case that rule *initialSchedule* creates a situation where not all remaining presentations can be scheduled in a way satisfying all conditions. The detail view of

¹⁴ Note that one pair in this set may indicate more than one conflict potentially occurring between the corresponding rules.

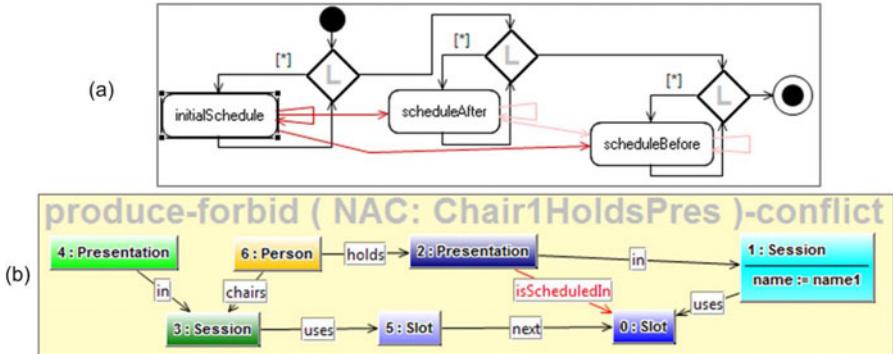


Fig. 5. (a) Potential conflicts along control flow caused by rule *initialSchedule*; (b) Detail view of potential conflict of rule *initialSchedule* with rule *scheduleAfter*

potential conflicts for pair $(initialSchedule, scheduleAfter)$ in Figure 5 (b) shows e.g. a potential produce-forbid conflict where rule *initialSchedule* (Figure 2) produces an edge from 2:Pres to 0:Slot, and rule *scheduleAfter* then must not schedule 4:Pres to 0:Slot because of the NAC shown in Figure 3.

4.4 Inspecting Trigger Causalities against Control Flow Direction

If rule a may trigger rule b and b is performed before a , then it might be the case that their order should be switched in order to obtain a complete simulation run. Given an integrated behavior model A with initial state S , this *plausibility check* automatically computes for each rule a in A , which successor rules of a may trigger a . The modeler then may inspect for each rule a in A that no rule performed after a exists that needs to be switched to a position before a in order to trigger its application (*desired property*). We identify the *critical signs* as the set of potential causalities against control flow triggered by a : $CrTrAg_A(a) = \{(a, b) | a, b \text{ rules in } A \text{ and } (a, b) \in ACFR_A \text{ such that } a \text{ may trigger } b\}$. We say that $CrTrAg_A = \{CrTrAg_A(a) | a \text{ is a rule in } A\}$ is the set of potential trigger causalities against control flow in A . We identify the *favorable signs* as the set of rules not triggered against control flow: $FaNoTrAg_A = \{a | a \text{ is rule in } A \text{ and } \nexists(b, a) \in CrTrAg_A\}$.

ACTIGRA visualizes the result of this plausibility check by displaying a *dashed red arrow* from a selected rule a to b for each pair of rules (a, b) in $CrTrAg_A(a)$. If no rule in particular is selected, then all pairs in $CrTrAg_A$ are displayed by dashed red arrows. Clicking on such an arrow from a to b opens a detail view, showing the reason(s) why a may trigger b as discovered by CSA. Conversely, each rule belonging to $FaNoTrAg_A$ is highlighted in green.

Example 5. In activity diagram *GenConfPlans* in Figure 6, we get the set of potential causalities against control flow $CrTrAg_{GenConfPlans}(createSession) = \{(createSession, Person2Pres)\}$. The causality $(createSession, Person2Pres)$

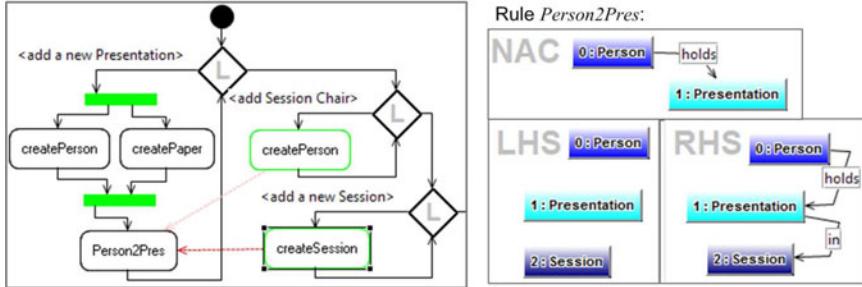


Fig. 6. Trigger causality against control flow (*createSession*, *Person2Pres*)

indicates that rule *Person2Pres* might be modelled too early in the control flow since rule *createSession* is needed to trigger rule *Person2Pres* completely.

4.5 Inspecting Causalities in Fork/Joins

We may not only consider the consistent sequential composition of rules as before, but consider also the parallel application of rules as specified by fork/join activities. Whenever a rule pair (a, b) belonging to the same fork/join may be causally dependent, then it is not possible to change their application order in any situation without changing the result. However, the parallel application of rules (a, b) implies that their application order should not matter.

Given an integrated behavior model A with initial state S , this *plausibility check* computes automatically for each fork/join in A , if potential causalities between the enclosed simple activities exist. The modeler may inspect each fork/join for its parallel execution not to be disturbed then (*desired property*).

We need some more elaborated considerations for this case, since we wish to analyze simple activities within a fork/join block that are normally disregarded as they only occur in the form of the corresponding parallel rule in $Sem(A)$. In particular, we define a *fork/join relation* FJR_A consisting of all rule pairs (a, b) belonging to the same fork/join block. We identify the *critical signs* as the set of potential causalities between different fork/join branches: $CrFJCa_A = \{(a, b) | (a, b) \in FJR_A \text{ and } (a, b) \text{ causally dependent}\}$.¹⁵ We identify the *favorable signs* as the set of fork/join structures with independent branches: $FaFJNoCa_A = \{fj | fj \text{ is fork/join in } A \text{ and } (a, b) \notin CrFJCa_A \text{ for each } (a, b) \text{ with } a, b \text{ in different branches of } fj\}$.

ACTIGRA visualizes the result of this plausibility check by displaying in each fork/join block a *dashed red arrow* from a to b for each $(a, b) \in CrFJCa_A$. The detail view shows the reason(s) why (a, b) are causally dependent and why this dependency might disturb parallel execution. On the other hand, each fork/join in $FaFJNoCa_A$ is highlighted by *green fork and join bars*.

¹⁵ Here, we do not only regard trigger causalities between a and b , but also causalities making the application of rule a irreversible as described in [13].



Fig. 7. Potential causality between different fork/join branches and its detail view

Example 6. The set of potential causalities between different fork/join branches depicted in Figure 7 is given by $\{(createPerson, Person2Pres)\}$. We may have a dependency (shown in the detail view) if rule *createPerson* creates a *Person* node that is used by rule *Person2Pres* to link it to a *Presentation* node.

5 Related Work

Our approach complements existing approaches that give a denotational semantics to activity diagrams by formal models. This semantics is used for validation purposes thereafter. For example, Eshuis [5] proposes a denotational semantics for a restricted class of activity models by means of labeled transition systems. Model checking is used to check properties. Störrle [18] defines a denotational semantics for the control flow of UML 2.0 activity models including procedure calls by means of Petri nets. The standard Petri net theory provides an analysis of properties like reachability or deadlock freeness. Both works stick to simple activities not further refined. In [3], business process models and web services are equipped with a combined graph transformation semantics and consistency can be validated by the model checker GROOVE. In contrast, we take integrated behavior models and check for potential conflict and causality inconsistencies between activity-specifying rules directly. Thus, our technique is not a “push-button” technique which checks a temporal formula specifying a desired property, but offers additional views on activity models where users can conveniently investigate intended and unintended conflicts and causalities between activities.

Fujaba [6], VMTS¹⁶ and GReAT¹⁷ are graph transformation tools for specifying and applying graph transformation rules along a control flow specified by activity models. However, controlled rule applications are not further validated concerning conflict and causality inconsistencies within these tools. Conflicts and causalities of pairs of rule-specified activities have been considered in various application contexts such as use case integration [8], feature modeling [9], model inconsistency detection [15], and aspect-oriented modeling [14]. Although sometimes embedded in explicit control flow, it has not been taken into account for inconsistency analysis.

¹⁶ Visual Modeling and Transformation System: <http://vmts.aut.bme.hu/>

¹⁷ Graph Rewriting and Transformation:

<http://www.isis.vanderbilt.edu/tools/great>

6 Conclusions and Future Work

Activity models are a wide-spread modeling technique to specify behavioral aspects of (software) systems. Here, we consider activity models where activities are integrated with object rules which describe pre- and post-conditions of activities based on a structural model. These integrated behavior models are formalized on the basis of graph transformation. The integrated specification of object rules within a control flow offers the possibility to find out potential conflict and causality inconsistencies. Actually, we can check if the order of rule applications specified by the control flow is plausible w.r.t. inherent potential conflicts and causalities of object rules. The Eclipse plug-in ACTIGRA prototypically implements these plausibility checks and visualizes potential conflicts and causalities in different views. Please note that our approach to plausibility reasoning can easily be adapted to any other approach where modeling techniques describing the control flow of operations, are integrated with operational rules like e.g. the integration of live sequence charts with object rules in [12].

A further refinement step in activity-based behavior modeling would be the specification of object flow between activities. Additionally specified object flow between two activities would further determine their inter-relation. In this case, previously determined potential conflicts and causalities might not occur anymore. Thus, the plausibility checks would become more exact with additionally specified object flow. A first formalization of integrated behavior models with object flow based on graph transformation is presented in [11]. An extension of plausibility checks to this kind of activity models is left for future work. Moreover, we plan to implement and visualize the sufficient criteria for consistency [10] in ACTIGRA. To conclude, integrated behavior models head towards a better integration of structural and behavioral modeling of (software) systems. Plausibility checks provide light-weight static analysis checks supporting the developer in constructing consistent models. Additionally, they allow modelers to reason about the necessity of sequencing activities.

References

1. Biermann, E., Ermel, C., Lambers, L., Prange, U., Taentzer, G.: Introduction to AGG and EMF Tiger by modeling a conference scheduling system. *Int. Journal on Software Tools for Technology Transfer* 12(3-4), 245–261 (2010)
2. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. *EATCS Monographs in Theor. Comp. Science*. Springer, Heidelberg (2006)
3. Engels, G., Güldali, B., Soltenborn, C., Wehrheim, H.: Assuring consistency of business process models and web services using visual contracts. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) *AGTIVE 2007. LNCS*, vol. 5088, pp. 17–31. Springer, Heidelberg (2008)
4. Ermel, C., Gall, J., Lambers, L., Taentzer, G.: Modeling with plausibility checking: Inspecting favorable and critical signs for consistency between control flow and functional behavior. *Tech. Rep. 2011/2, TU Berlin* (2011), <http://www.eecs.tu-berlin.de/menue/forschung/forschungsberichte/>

5. Eshuis, R., Wieringa, R.: Tool support for verifying UML activity diagrams. *IEEE Transactions on Software Engineering* 7(30) (2004)
6. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000)
7. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science* 19, 1–52 (2009)
8. Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In: Proc. ICSE, pp. 105–115. ACM, New York (2002)
9. Jayaraman, P.K., Whittle, J., Elkhodary, A.M., Gomaa, H.: Model composition in product lines and feature interaction detection using critical pair analysis. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 151–165. Springer, Heidelberg (2007)
10. Jurack, S., Lambers, L., Mehner, K., Taentzer, G.: Sufficient Criteria for Consistent Behavior Modeling with Refined Activity Diagrams. In: Czarnecki, K. (ed.) MODELS 2008. LNCS, vol. 5301, pp. 341–355. Springer, Heidelberg (2008)
11. Jurack, S., Lambers, L., Mehner, K., Taentzer, G., Wierse, G.: Object Flow Definition for Refined Activity Diagrams. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 49–63. Springer, Heidelberg (2009)
12. Lambers, L., Mariani, L., Ehrig, H., Pezze, M.: A Formal Framework for Developing Adaptable Service-Based Applications. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 392–406. Springer, Heidelberg (2008)
13. Lambers, L.: Certifying Rule-Based Models using Graph Transformation. Ph.D. thesis, Technische Universität Berlin (2009)
14. Mehner, K., Monga, M., Taentzer, G.: Analysis of aspect-oriented model weaving. In: Rashid, A., Ossher, H. (eds.) Transactions on AOSD V. LNCS, vol. 5490, pp. 235–263. Springer, Heidelberg (2009)
15. Mens, T., Straeten, R.V.D., D'Hondt, M.: Detecting and resolving model inconsistencies using transformation dependency analysis. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 200–214. Springer, Heidelberg (2006)
16. Object Management Group. Unified Modeling Language: Superstructure – Version 2.3 (2010), <http://www.omg.org/spec/UML/2.3/> formal/07-02-05
17. Rensink, A., Van Gorp, P. (eds.): Int. Journal on Software Tools for Technology Transfer, Section on Graph Transf. Tool Contest 2008, vol. 12(3-4). Springer, Heidelberg (2010)
18. Störrle, H.: Semantics of UML 2.0 activity diagrams. In: VLHCC 2004. IEEE, Los Alamitos (2004)

Models within Models: Taming Model Complexity Using the Sub-model Lattice

Pierre Kelsen, Qin Ma, and Christian Glodt

Laboratory for Advanced Software Systems

University of Luxembourg

6, rue Richard Coudenhove-Kalergi

L-1359 Luxembourg

{Pierre.Kelsen, Qin.Ma, Christian.Glodt}@uni.lu

Abstract. Model-driven software development aims at easing the process of software development by using models as primary artifacts. Although less complex than the real systems they are based on, models tend to be complex nevertheless, thus making the task of comprehending them non-trivial in many cases. In this paper we propose a technique for model comprehension based on decomposing models into sub-models that conform to the same metamodel as the original model. The main contributions of this paper are: a mathematical description of the structure of these sub-models as a lattice, a linear-time algorithm for constructing this decomposition and finally an application of our decomposition technique to model comprehension.

1 Introduction

In model-driven software development models are the primary artifacts. Typically several models are used to describe the different concerns of a system. One of the main motivations for using models is the problem of dealing with the complexity of real systems: because models represent abstractions of a system, they are typically less complex than the systems they represent.

Nevertheless models for real systems can be complex themselves and thus may require aids for facilitating human comprehension. The problem of understanding complex models is at the heart of this paper. We propose a method for decomposing models that is based on subdividing models into smaller sub-models with the property that these sub-models conform to the same metamodel as the original model. This property allows to view the sub-models using the same tools as the original model and to understand the meaning of the sub-models using the same semantic mapping (if one has been defined).

An example of a concrete application scenario is the following: when trying to understand a large model, one starts with a subset of concepts that one is interested in (such as the concept of Class in the UML metamodel). Our method allows to construct a small sub-model of the initial model that contains all entities of interest and that conforms to the original metamodel (in the case of UML this would be MOF). The latter condition ensures that the sub-model can

be viewed in the same way as the original model and that it has a well-defined semantics. The smaller size (compared to the original model) should facilitate comprehension.

The main contributions of this paper are the following: (a) we present the mathematical structure of these sub-models as a lattice, with the original model at the top and the empty sub-model at the bottom; (b) we present a linear time algorithm for building a decomposition hierarchy for a model from which the sub-model lattice can be constructed in a straightforward manner; (c) we present a method for the user to comprehend models in the context of model pruning.

One salient feature of our technique is its generic nature: it applies to any model (even outside the realm of model-driven software development). This is also one of the main features differentiating it from existing work in model decomposition. We review here related work from model slicing, metamodel pruning and model abstraction.

The idea behind model slicing is to generalize the work on program slicing to the domains of models by computing parts of models that contain modeling elements of interest. An example of this line of work is [4] where model slicing of UML class diagrams is investigated. Another example is [2] which considers the problem of slicing the UML metamodel into metamodels corresponding to the different diagram types in UML. The main differences between our work and research on model slicing is, first, the restriction of model slicing to a particular modeling language, e.g. UML class diagrams, and, second, the focus on a single model rather than the mathematical structure of sub-models of interest.

In a similar line of work some authors have investigated the possibility of pruning metamodels in order to make them more manageable. The idea is to remove elements from a metamodel to obtain a minimal set of modeling elements containing a given subset of elements of interest. Such an approach is described in [8]. This work differs from our work in several respects: first, just like model slicing it focuses on a single model rather than considering the collection of relevant sub-models in its totality; second, it is less generic in the sense that it restricts its attention to Ecore metamodels (and the pruning algorithm they present is very dependent on the structure of Ecore), and lastly their goal is not just to get a conformant sub-model but rather to find a sub-metamodel that is a supertype of the original model. This added constraint is due to main use of the sub-metamodel in model transformation testing.

The general idea of simplifying models (which can be seen as a generalization of model slicing and pruning) has also been investigated in the area of model abstraction (see [3] for an overview). In the area of simulation model abstraction is a method for reducing the complexity of a simulation model while maintaining the validity of the simulation results with respect to the question that the simulation is being used to address. Work in this area differs from ours in two ways: first, model abstraction techniques generally transform models and do not necessarily result in sub-models; second, conformance of the resulting model with a metamodel is not the main concern but rather validity of simulation results.

The remainder of this paper is structured as follows: in the next section we present formal definitions for models, metamodels and model conformance. In section 3 we describe an algorithm for decomposing a model into sub-models conformant to the same metamodel as the original model. We also present the mathematical structure of these models, namely the lattice of sub-models. In section 4 we outline the application to model comprehension and we present concluding remarks in the final section.

2 Models and MetaModels

In this section we present formal definitions of models, metamodels, and model conformance. The following notational conventions will be used:

1. For any tuple p , we use $\text{fst}(p)$ to denote its first element, and $\text{snd}(p)$ to denote its second element.
2. For any set s , we use $\#s$ to denote its cardinality.
3. We use \leq to denote the inheritance based subtyping relation.

2.1 Metamodels

A metamodel defines (the abstract syntax of) a language for expressing models. It consists of a finite set of metaclasses and a finite set of relations between metaclasses - either associations or inheritance relations. Moreover, a set of constraints may be specified in the contexts of metaclasses as additional well-formedness rules.

Definition 1 (Metamodel). A metamodel $\mathbb{M} = (\mathcal{C}, \mathcal{A}, \mathcal{R})$ is a tuple:

- \mathcal{C} is the set of metaclasses, and $\in \mathcal{C}$ ranges over it.
- $\subseteq (\mathcal{C} \times \mu) \times (\mathcal{C} \times \mu \times \mathcal{C})$ represents the (directed) associations between metaclasses and $\in \mathcal{C}$ ranges over it. The two \mathcal{C} 's give the types of the two association ends. The two μ 's, where $\mu \in \text{Int} \times \{\text{Int} \cup \{\infty\}\}$, give the corresponding multiplicities. We refer to the first end of the association as the source, and the second as the target. Associations are navigable from source to target, and the navigation is represented by referring to the given role name that is attached to the target end selected from the vocabulary of role names.
- $\subseteq \mathcal{C} \times \mathcal{C}$ denotes the inheritance relation among metaclasses and $\in \mathcal{C}$ ranges over it. For a given $\in \mathcal{C}$, $\text{fst}(\mathcal{C})$ inherits from (i.e., is a subtype of) $\text{snd}(\mathcal{C})$.
- $\subseteq \mathcal{C} \times \mathbb{E}$ gives the set of constraints applied to the metamodel and $\in \mathcal{C}$ ranges over it. \mathbb{E} is the set of the expressions and $\in \mathbb{E}$ ranges over it. A constraint $= (\mathcal{C}, \mathbb{E})$ makes the context metaclass \mathcal{C} more precise by restricting it with an assertion, i.e., the boolean typed expression \mathbb{E} . For example, a constraint can further restrict the multiplicities or types of association ends that are related to the context metaclass.

Let $\in \mathcal{C}$ range over the set of role names mentioned above. We require that role names in a metamodel are distinct. As a consequence, it is always possible to retrieve the association that corresponds to a given role name, written $\text{asso}(\mathcal{C})$.

2.2 Models

A model is expressed in a metamodel. It is built by instantiating the constructs, i.e., metaclasses and associations, of the metamodel.

Definition 2 (Model). *A model is defined by a tuple $M = (M, N, A, \tau)$ where:*

- M is the metamodel in which the model is expressed.
- N is the set of metaclass instantiations of the metamodel M , and $n \in N$ ranges over it. They are often simply referred to as instances when there is no possible confusion.
- $A \subseteq N \times (N \times \cdot)$ is the set of association instantiations of the metamodel M , and $a \in A$ ranges over it. They are often referred to as links.
- τ is the typing function: $(N \rightarrow \cdot) \cup (A \rightarrow \cdot)$. It records the type information of the instances and links in the model, i.e., from which metaclasses or associations of the metamodel M they are instantiated.

2.3 Model Conformance

Not all models following the definitions above are valid, or “conform to” the metamodel: typing, multiplicity, and constraints need all to be respected.

Definition 3 (Model conformance). *We say a model $M = (M, N, A, \tau)$ conforms to its metamodel M or is valid when the following conditions are met:*

1. *type compatible:*

$$\forall a \in A, \tau(\text{fst}(a)) \leq \text{fst}(\text{fst}(\tau(a))) \text{ and } \tau(\text{fst}(\text{snd}(a))) \leq \text{fst}(\text{snd}(\tau(a)))$$

Namely, the types of the link ends must be compatible with (being subtypes of) the types as specified in the corresponding association ends.

2. *multiplicity compatible:* $\forall n \in N, \cdot \in \cdot$,

if $\tau(n) \leq \text{fst}(\text{fst}(\cdot))$,

then $\#\{a \mid a \in A \text{ and } \tau(a) = \cdot \text{ and } \text{fst}(a) = n\} \in \text{snd}(\text{snd}(\cdot))$;

if $\tau(n) \leq \text{fst}(\text{snd}(\cdot))$,

then $\#\{a \mid a \in A \text{ and } \tau(a) = \cdot \text{ and } \text{fst}(\text{snd}(a)) = n\} \in \text{snd}(\text{fst}(\cdot))$.

Namely, the number of link ends should conform to the specified multiplicity in the corresponding association end.

3. *constraints hold:* $\forall \cdot \in \cdot, \forall n \in N$ where n is an instance of the context metaclass, i.e., $\tau(n) \leq \text{fst}(\cdot)$, the boolean expression $\text{snd}(\cdot)$ should evaluate to true in model M for the contextual instance n .

3 Model Decomposition

3.1 Criteria

Model decomposition starts from a model that conforms to a metamodel, and decomposes it into smaller parts. Our model decomposition technique is designed using the following as main criterion: the derived parts should be valid models conforming to the original metamodel. Achieving this goal has two main advantages:

1. the derived parts, being themselves valid models, can be comprehended on their own according to the familiar abstract syntax and semantics (if defined) of the modeling language;
2. the derived parts can be wrapped up into modules and reused in the construction of other system models, following our modular model composition paradigm [6].

The decomposed smaller parts of a model are called its *sub-models*, formally defined below.

Definition 4 (Sub-model). *We say a model $M' = (\mathbb{M}, N', A', \tau')$ is a sub-model of another model $M = (\mathbb{M}, N, A, \tau)$ if and only if:*

1. $N' \subseteq N$;
2. $A' \subseteq A$;
3. τ' is a restriction of τ to N' and A' .

In order to make the sub-model M' also conform to \mathbb{M} , we will propose three conditions - one for the metamodel (Condition 3 below, regarding the nature of the constraints) and two conditions for the sub-model (Conditions 1 and 2). Altogether these three conditions will be sufficient to ensure conformance of the sub-model.

The starting point of our investigation is the definition of conformance (Definition 3). Three conditions must be met in order for sub-model M' to conform to metamodel \mathbb{M} .

The first condition for conformance, type compatibility, follows directly from the fact that M' is a sub-model of M and M conforms to \mathbb{M} . The second condition for conformance, which we call the *multiplicity condition*, concerns the multiplicities on the association ends in the metamodel \mathbb{M} . First the number of links ending at an instance of M' must agree with the source cardinality of the corresponding association in the metamodel and second the number of links leaving an instance of M' must agree with the target cardinality of the corresponding association.

To ensure the multiplicity condition for links ending at an instance of the sub-model, we will introduce the notion of *fragmentable links*, whose type (i.e., the corresponding association) has an un-constrained (i.e., being 0) lower bound for the source cardinality.

Definition 5 (Fragmentable link). *Given a model $M = (\mathbb{M}, N, A, \tau)$, a link $a \in A$ is fragmentable if $\text{snd}(\text{fst}(\tau(a))) = (0, -)$, where $-$ represents any integer whose value is irrelevant for this definition.*

Fragmentable incoming links of M to instances in M' are safe to exclude but this is not the case for non-fragmentable links, which should all be included. We thus obtain the first condition on sub-model M' :

Condition 1. $\forall a \in A \text{ where } a \text{ is non-fragmentable, } \text{fst}(\text{snd}(a)) \in N' \text{ implies } \text{fst}(a) \in N' \text{ and } a \in A'$.

Let us now consider the multiplicity condition for links leaving an instance of M' . To ensure this condition we shall require that M' includes all the links of M that leave an instance of M' . In other words, there are in fact no links leaving M' . This is formally expressed in the following condition on the sub-model:

Condition 2. $\forall a \in A, \text{fst}(a) \in N' \text{ implies } \text{fst}(\text{snd}(a)) \in N' \text{ and } a \in A'$.

Conditions 1 and 2 together imply the multiplicity condition in the conformance definition.

The third condition in the conformance definition, which we call the *constraint condition*, requires that all metamodel constraints are satisfied in sub-model M' . To ensure this condition, we impose a restriction on the nature of the constraints. To this end, we introduce the notion of forward constraint.

Definition 6 (Forward constraint). *A constraint is forward if for any model M and any instance n of M , the instances and links referenced by constraint with contextual instance n are reachable from n in M (viewed as a directed graph).*

We will only consider forward constraints in this paper. This is expressed in the following condition over the metamodel M :

Condition 3. *All constraints in metamodel M are forward constraints.*

Note that we have formalized a core part of the EssentialOCL [7] in the companion technical report [5], which in principle excludes `AllInstances`, called CoreOCL and we prove in [5] that all CoreOCL constraints are forward.

It is not difficult to see that both Conditions 2 and 3 imply the constraint condition in the conformance definition. Indeed Condition 2 implies that all instances and links reachable from an instance n in model M are also reachable in M' if M' does indeed contain n . Condition 3 then implies that a constraint that is satisfied on contextual instance n in M is also satisfied in the sub-model M' since it references the same instances and links in both models.

We thus obtain the following result:

Theorem 1. *Given a metamodel M , a model M , and a sub-model M' of M , suppose that:*

1. *model M conforms to M ;*
2. *model M' satisfies Condition 1 and 2;*
3. *metamodel M satisfies Condition 3;*

then model M' also conforms to the metamodel M .

Proof. The result follows from the discussion above.

3.2 Algorithm

From hereon we shall assume that the metamodel under consideration satisfies Condition 3. In this subsection we describe an algorithm that finds, for a given model M , a decomposition of M such that any sub-model of M that satisfies both Condition 1 and 2 can be derived from the decomposition by uniting some components of the decomposition.

We reach the goal in two steps: (1) ensure Condition 1 and 2 with respect to only non-fragmentable links; (2) ensure again Condition 2 with respect to fragmentable links. (Condition 1 does not need to be re-assured because it only involves non-fragmentable links.) Details of each step are discussed below.

Treating instances as vertices and links as edges, models are just graphs. For illustration purpose, consider an example model as presented in Figure 1 where all fragmentable links are indicated by two short parallel lines crossing the links.

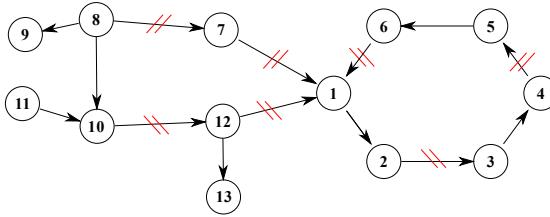


Fig. 1. An example model

Let G be the graph derived by removing the fragmentable links from M . Because all the links in G are non-fragmentable, for a sub-graph of G to satisfy both Condition 1 and 2, an instance is included in the sub-graph if and only if all its ancestor and descendant instances are also included, and so are the links among these instances in G . These instances, from the point of view of graph theory, constitute a weakly connected component (wcc) of graph G (i.e., a connected component if we ignore edge directions). The first step of the model decomposition computes all such wcc's of G , which disjointly cover all the instances in model M , then puts back the fragmentable links. We collapse all the nodes that belong to one wcc into one node, (referred to as a *wcc-node* in contrast to the original nodes), and refer to the result as graph W . After the first step, the corresponding graph W of the example model contains six wcc-nodes inter-connected by fragmentable links, as shown in Figure 2.

The instances and links that are collapsed into one wcc-node in W constitute a sub-model of M satisfying both Condition 1 and 2, but only with respect to non-fragmentable links, because wcc's are computed in the context of G where fragmentable links are removed. The second step of the model decomposition starts from graph W and tries to satisfy Condition 2 with respect to fragmentable links, i.e., following outgoing fragmentable links. More specifically, we compute all the strongly connected components (scc's) in W (see [10] for a definition of strongly connected components) and collapse all the nodes that belong to one

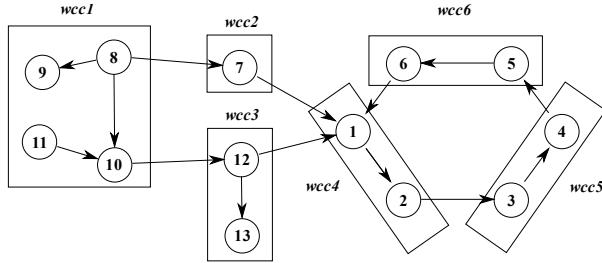


Fig. 2. The corresponding W graph of the example model after step 1

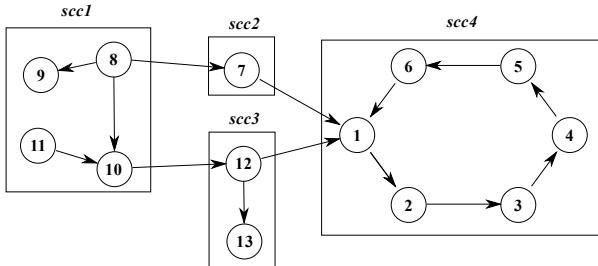


Fig. 3. The corresponding D graph of the example model after step 2

scc into one node, (referred to as *an scc-node*), and refer to the result as graph D . After the second step, the corresponding graph D of the example model looks like in Figure 3. The three wcc-nodes $wcc4$, $wcc5$ and $wcc6$ of graph W are collapsed into one scc-node $scc4$ because they lie on a (directed) cycle.

Note that we only collapse nodes of a strongly connected component in the second step instead of any reachable nodes following outgoing fragmentable links in W , because we do not want to loose any potential sub-model of M satisfying both Condition 1 and 2 on the way. More precisely, a set of nodes is collapsed only if for every sub-model M' of M satisfying both Condition 1 and 2, it is either completely contained in M' or disjoint with M' , i.e., no such M' can tell the nodes in the set apart.

The computational complexity of the above algorithm is dominated by the complexity of computing weakly and strongly connected components in the model graph. Computing weakly connected components amounts to computing connected components if we ignore the direction of the edges. We can compute connected components and strongly connected components in linear time using depth-first search [10]. Thus the overall complexity is linear in the size of the model graph.

3.3 Correctness

Graph D obtained at the end of the algorithm is a DAG (Directed Acyclic Graph) with all the edges being fragmentable links. Graph D represents a decomposition

of the original model M where all the instances and links that are collapsed into an scc-node in D constitute a component in the decomposition. We call graph D the *decomposition hierarchy* of model M .

To relate the decomposition hierarchy to the sub-models, we introduce the concept of an *antichain-node*. An antichain-node is derived by collapsing a (possibly empty) antichain of scc-nodes (i.e., a set of scc-nodes that are neither descendants nor ancestors of one another, the concept of antichain being borrowed from order theory) plus their descendants (briefly an antichain plus descendants) in the decomposition hierarchy. To demonstrate the correctness of the algorithm, we prove the following theorem:

Theorem 2. *Given a model $M = (\mathbb{M}, \mathbb{N}, \mathbb{A}, \tau)$ and a sub-model $M' = (\mathbb{M}', \mathbb{N}', \mathbb{A}', \tau)$ of M , M' satisfies both Condition 1 and 2 if and only if there exists a corresponding antichain-node of the decomposition hierarchy of M where M' consists of the instances and links collapsed in this antichain-node.*

Proof. We first demonstrate that if M' consists of the set of instances and links that are collapsed in an antichain-node of the decomposition hierarchy of M , then M' satisfies both Condition 1 and 2.

- Check M' against Condition 1: given a non-fragmentable link $a \in A$, if $\text{fst}(\text{snd}(a)) \in N'$, we have $\text{fst}(a) \in N'$ because of the wcc computation in the first step of the model decomposition algorithm.
- Check M' against Condition 2: given a non-fragmentable link $a \in A$, if $\text{fst}(a) \in N'$, we have $\text{fst}(\text{snd}(a)) \in N'$ because of the wcc computation in the first step of the model decomposition algorithm. Given a fragmentable link $a \in A$, if $\text{fst}(a) \in N'$, we have $\text{fst}(\text{snd}(a)) \in N'$ because of the scc computation in the second step of the model decomposition algorithm and because we take all the descendants into account.

We now demonstrate the other direction of the theorem, namely, if M' satisfies both Condition 1 and 2, then there exists an antichain-node of the decomposition hierarchy of M , such that M' consists of the set of instances and links that are collapsed in this antichain-node.

We refer to the set of scc-nodes in the decomposition hierarchy where each includes at least one instance of M' by S .

1. All the instances that are collapsed in an scc-node in S belong to M' . Given an scc-node $s \in S$, there must exist an instance n collapsed in s and $n \in N'$ in order for s to be included in S . Let n' be another instance collapsed in s . Following the algorithm in Section 3.2 to compute the decomposition hierarchy, n and n' are aggregated into one scc-node either in the first or the second step.
 - (a) If they are aggregated in the first step, that means the two instances are weakly connected by non-fragmentable links, and because M' satisfies Condition 1 and 2, n' should also be in M' .
 - (b) If they are aggregated in the second step but not in the first step, that means n and n' are aggregated in two separate wcc-nodes in the first step,

called w and w' , which are strongly connected by a path of fragmentable links. Referring to the other wcc-nodes on the path by w_1, \dots, w_k , there exists a set of instances $n_0 \in w$, $n'_0 \in w'$, and $n_i, n'_i \in w_i$ for $1 \leq i \leq k$, such that there are fragmentable links from n_0 to n_1 , from n'_i to n_{i+1} ($\forall i. 1 \leq i < k$) and from n'_k to n'_0 . Since M' satisfies Condition 2 if the source vertex of these fragmentable links belongs to M' , so does the target vertex. Because the following pairs of instances: n and n_0 , n_i and n'_i ($\forall i. 1 \leq i \leq k$), and n'_0 and n' , are respectively collapsed in a wcc-node, if one vertex in a pair belongs to M' then the other vertex in the pair must belong to M' as well following Condition 1 and 2. From the above discussion and by applying mathematical induction, n belonging to M' implies that n' belongs to M' as well.

2. S constitutes an antichain plus descendant. We partition S into two subsets: S_1 contains all the scc-nodes in S that do not have another scc-node also in S as ancestor; S_2 contains the rest, i.e., $S_2 = S \setminus S_1$. Clearly S_1 constitutes an antichain, and any scc-node in S_2 is a descendant of an scc-node in S_1 because otherwise the former scc-node should belong to S_1 instead of S_2 . Moreover, S_2 contains all the descendants of scc-nodes in S_1 . Given a child s_2 of an scc-node $s_1 \in S_1$, the fragmentable link from s_1 to s_2 connects an instance n_1 collapsed in s_1 to an instance n_2 collapsed in s_2 . Because $s_1 \in S_1$, following the demonstrated item 1 above, we have $n_1 \in N'$. Because of the out-going fragmentable link from n_1 to n_2 and since M' satisfies Condition 2, we also have $n_2 \in N'$. Therefore we have $s_2 \in S$. Furthermore, $s_2 \notin S_1$ because it has $s_1 \in S$ as its ancestor. Hence we have $s_2 \in S_2$. Inductively we conclude that any descendant of s_1 belongs to S_2 and hence S constitutes an antichain plus descendant.
3. Collapse all the scc-nodes in S into an antichain-node called A . We demonstrate that M' consists of the instances and links collapsed in A .
 - (a) Any instance of M' is collapsed in A because of the selection criteria of S , and any instance collapsed in A is an instance of M' following the demonstrated item 1 above. In other words, M' and A contain the same set of instances from M .
 - (b) Because both M' and A span all the links in M that connect instances in them, M' and A also have the same set of links from M .

3.4 The Lattice of Sub-models

Recall that a lattice is a partially-ordered set in which every pair of elements has a least upper bound and a greatest lower bound. Thanks to Theorem 2, we can now refer to a sub-model M' of model M that satisfies both Condition 1 and 2 by the corresponding antichain-node A in the decomposition hierarchy of M . Given a model M , all the sub-models that satisfy both Condition 1 and 2 constitute a lattice ordered by the relation “is a sub-model of”, referred to as *the sub-model lattice* of M . Let A_1 and A_2 denote two such sub-models. The least upper bound ($A_1 \vee A_2$) and the greatest lower bound ($A_1 \wedge A_2$) of A_1 and A_2 are computed in the following way:

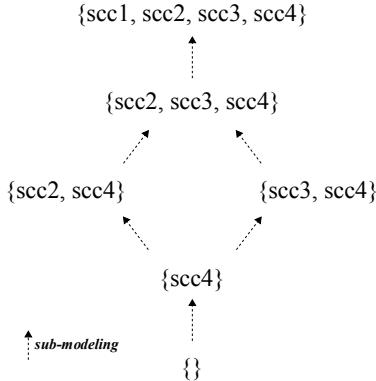


Fig. 4. The sub-model lattice of the example model in Figure 1 whose decomposition hierarchy is given in Figure 2

- $A_1 \vee A_2$ is the antichain-node obtained by collapsing the scc-nodes of A_1 and A_2 ;
- $A_1 \wedge A_2$ is the antichain-node obtained by collapsing the common scc-nodes of A_1 and A_2 .

The top of the sub-model lattice is M itself, and the bottom is the empty sub-model.

For the example model discussed in Section 3.2 whose decomposition hierarchy is given in Figure 3, six possible antichain-nodes can be derived from the decomposition hierarchy, denoted by the set of scc-nodes that are collapsed. They are ordered in a lattice as shown in Figure 4.

3.5 Implementation

We have implemented the model decomposition technique [1]. The implementation takes a model of any metamodel that follows Definition 1 as input, and computes the decomposition hierarchy of it from which the sub-model lattice can be constructed by enumerating all the antichain-nodes of the decomposition hierarchy. Note that in the worst case where the decomposition hierarchy contains no edges, the size of the sub-model lattice equals the size of the power-set of the decomposition hierarchy, which is exponential.

4 Application: Pruning Based Model Comprehension

In this section, we demonstrate the power of our generic model decomposition technique by reporting one of its applications in a pruning-based model comprehension method. A typical comprehension question one would like to have answered for a large model is:

Given a set of instances of interest in the model, how does one construct a substantially smaller sub-model that is relevant for the comprehension of these instances?

Model readers, when confronted with such a problem, would typically start from the interesting instances and browse through the whole model attempting to manually identify the relevant parts. Even with the best model documentation and the support of model browsing tools, such a task may still be too complicated to solve by hand, especially when the complexity of the original model is high. Moreover, guaranteeing by construction that the identified parts (together with the interesting instances) indeed constitute a valid model further complicates the problem.

Our model decomposition technique can be exploited to provide a linear time automated solution to the problem above. The idea is to simply take the union of all the scc-nodes, each of which contains at least one interesting instance, and their descendant scc-nodes in the decomposition hierarchy of the original model. We have implemented the idea in a Ecore [9] model comprehension tool [1] based on the implementation of the model decomposition technique.

To assess the applicability of the tool, a case study has been carried out. We have chosen the Ecore model of BPMN (Business Process Modeling Notation) [11] `bpmn.ecore` as an example, and one of BPMN's main concepts – *Gateway* – for comprehension. Gateways are modeling elements in BPMN used to control how sequence flows interact as they converge and diverge within a business process. Five types of gateways are identified in order to cater to different types of sequence flow control semantics: exclusive, inclusive, parallel, complex, and event-based.

Inputs to the comprehension tool for the case study are the following:

- The BPMN Ecore model containing 134 classes (EClass instances), 252 properties (EReference instances), and 220 attributes (EAttribute instances). Altogether, it results in a very large class diagram that does not fit on a single page if one wants to be able to read the contents properly. Figure 5 shows the bird's eye view of this huge diagram.
- a set of interesting instances capturing the key notions of the design of gateways in BPMN: *Gateway*, *ExclusiveGateway*, *InclusiveGateway*, *ParallelGateway*, *ComplexGateway*, and *EventBasedGateway*.

After applying the tool, the result BPMN sub-model that contains all the selected interesting instances has only 17 classes, 7 properties, and 21 attributes. We observe that all the other independent concepts of BPMN such as *Activity*, *Event*, *Connector*, and *Artifact*, are pruned out. The class diagram view of the pruned BPMN model is shown in Figure 6. Note that it corresponds well to the class diagram that is sketched in the chapter for describing gateways in the BPMN 2.0 specification [11]. We have also verified that the pruned BPMN model is indeed an Ecore instance by validating it against `ecore.ecore` in EMF [9].

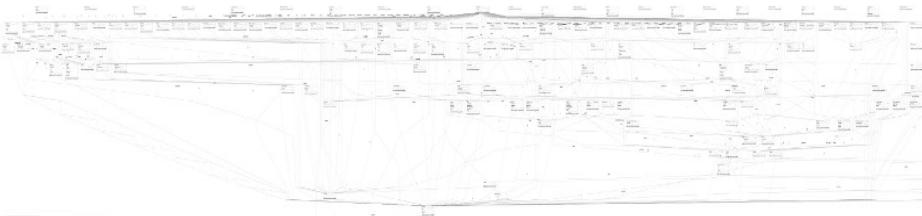


Fig. 5. Bird's Eye View of the Class diagram of the BPMN Ecore model

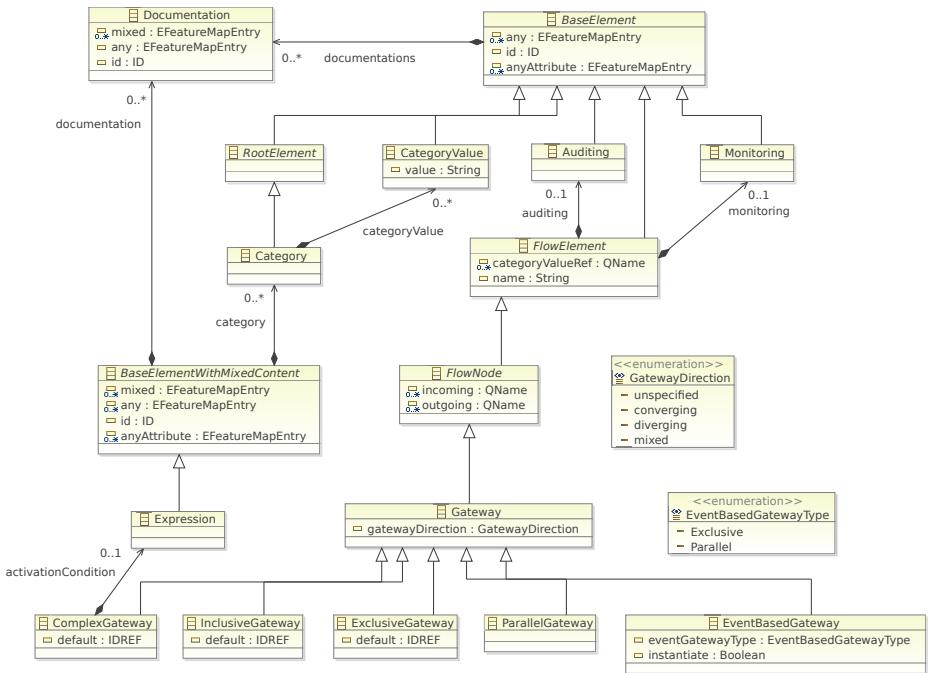


Fig. 6. Pruned class diagram for understanding the Gateway concept in BPMN

5 Conclusion and Future Work

The lattice of sub-models described in this paper should have applications beyond the application for model comprehension described in this paper. We foresee potential applications in the areas of model testing, debugging, and model reuse.

Given a software that takes models of a metamodel as input (e.g., a model transformation), an important part in testing the software is test case generation. Our model decomposition technique could help with the generation of new test

cases by using one existing test case as the seed. New test cases of various complexity degree could be automatically generated following the sub-model lattice of the seed test case.

Moreover, our model decomposition can also help with the debugging activity when a failure of the software is observed on a test case. The idea is that we will find a sub-model of the original test case which is responsible for triggering the bug. Although both the reduced test case and the original one are relevant, the smaller test case is easier to understand and investigate.

A major obstacle to the massive model reuse in model-based software engineering is the cost of building a repository of reusable model components. A more effective alternative to creating those reusable model components from scratch is to discover them from existing system models. Sub-models of a system extracted by following our model decomposition technique are all guaranteed to be valid models hence can be wrapped up into modules and reused in the construction of other systems following our modular model composition paradigm [6].

Our model decomposition technique, described in this paper can be further improved: indeed it is currently based on three sufficient conditions that are not necessary. A consequence of this is that not all conformant sub-models are captured in the lattice of sub-models. A finer analysis of the constraints in the metamodel could result in weakening the three conditions and thus provide a more complete collection of conformant sub-models.

Acknowledgment. We would like to thank the anonymous referees for making numerous comments that helped us in improving the presentation of this paper.

References

1. Democles tool, <http://democles.lassy.uni.lu/>
2. Bae, J.H., Lee, K., Chae, H.S.: Modularization of the UML metamodel using model slicing. In: Fifth International Conference on Information Technology: New Generations, pp. 1253–1254 (2008)
3. Frantz, F.K.: A taxonomy of model abstraction techniques. In: Winter Simulation Conference, pp. 1413–1420 (1995)
4. Kagdi, H., Maletic, J.I., Sutton, A.: Context-free slicing of UML class models. In: ICSM 2005: Proceedings of the 21st IEEE International Conference on Software Maintenance, pp. 635–638. IEEE Computer Society, Washington, DC, USA (2005)
5. Kelsen, P., Ma, Q.: A generic model decomposition technique. Technical Report TR-LASSY-10-06, Laboratory for Advanced Software Systems, University of Luxembourg (2010), http://democles.lassy.uni.lu/documentation/TR_LASSY_10_06.pdf
6. Kelsen, P., Ma, Q.: A modular model composition technique. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 173–187. Springer, Heidelberg (2010)
7. OMG. Object Constraint Language version 2.2 (February 2010)

8. Sen, S., Moha, N., Baudry, B., Jézéquel, J.-M.: Meta-model pruning. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 32–46. Springer, Heidelberg (2009)
9. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional, Reading (2008)
10. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. 1(2), 146–160 (1972)
11. White, S.A., Miers, D.: BPMN Modeling and Reference Guide. Future Strategies Inc. (2008)

Type-Safe Evolution of Spreadsheets

Jácome Cunha^{1,2,*}, Joost Visser³, Tiago Alves^{1,3, **}, and João Saraiva¹

¹ Universidade do Minho, Portugal

{jacome, jas}@di.uminho.pt

² ESTGF, Instituto Politécnico do Porto, Portugal

³ Software Improvement Group, The Netherlands

{j.visser, t.alves}@sig.eu

Abstract. Spreadsheets are notoriously error-prone. To help avoid the introduction of errors when changing spreadsheets, models that capture the structure and interdependencies of spreadsheets at a conceptual level have been proposed. Thus, spreadsheet evolution can be made safe within the confines of a model. As in any other model-instance setting, evolution may not only require changes at the instance level but also at the model level. When model changes are required, the safety of instance evolution can not be guarded by the model alone.

We have designed an appropriate representation of spreadsheet models, including the fundamental notions of formulæ and references. For these models and their instances, we have designed coupled transformation rules that cover specific spreadsheet evolution steps, such as the insertion of columns in all occurrences of a repeated block of cells. Each model-level transformation rule is coupled with instance level migration rules from the source to the target model and vice versa. These coupled rules can be composed to create compound transformations at the model level inducing compound transformations at the instance level. This approach guarantees safe evolution of spreadsheets even when models change.

1 Introduction

Spreadsheets are widely used by non-professional programmers, the so-called *end users*, to develop business applications. Spreadsheet systems offer end users a high level of flexibility, making it easier to get started working with them. This freedom, however, comes with a price: spreadsheets are error prone as shown by numerous studies which report that up to 90% of real-world spreadsheets contain errors [19,21,22].

As programming systems, spreadsheets lack the support provided by modern programming languages/environments, like for example, higher-level abstractions and powerful type and modular systems. As a result, they are prone to errors. In order to improve end-users productivity, several techniques have been recently proposed, which guide end users to safely/correctly edit spreadsheets, like, for example, the use of spreadsheet templates [2], ClassSheets [8,11], and the inclusion of visual objects to provide editing assistance in spreadsheets [10]. All these approaches propose a form of

* Supported by Fundação para a Ciência e a Tecnologia, grant no. SFRH/BD/30231/2006.

** Supported by Fundação para a Ciência e a Tecnologia, grant no. SFRH/BD/30215/2006.

Work supported by the SSaaPP project, FCT contract no. PTDC/EIA-CCO/108613/2008.

end user model-driven software development: a spreadsheet business model is defined, from which then a customized spreadsheet application is generated guaranteeing the consistency of the spreadsheet data with the underlying model. In a recent empirical study we have shown that the use of model-based spreadsheets do improve end-users productivity [7].

Despite of its huge benefits, model-driven software development is sometimes difficult to realize in practice due to two main reasons: first, as some studies suggest, defining the business model of a spreadsheet can be a complex task for end users [1]. As a result, they are unable to follow this spreadsheet development discipline. Second, things get even more complex when the spreadsheet model needs to be updated due to new requirements of the business model. End users need not only to evolve the model, but also to migrate the spreadsheet data so that it remains consistent with the model. To address the first problem, in [8] we have proposed a technique to derive the spreadsheet's business model, represented as a ClassSheet model, from the spreadsheet data. In this paper we address the second problem, that is, the co-evolution of the spreadsheet model and the spreadsheet data (*i.e.*, the instance of the model). Co-evolution of models and instances are supported by the two-level coupled transformation framework [4].

In this paper we present an appropriate representation of a spreadsheet model, based on the ClassSheet business model, including the fundamental notions of formulæ, references, and expandable blocks of cells. For this model and its instance, we design coupled transformation rules that cover specific spreadsheet evolution steps, such as extraction of a block of cells into a separate sheet or insertion of columns in all occurrences of a repeated block of cells. Each model-level transformation rule is coupled with instance level migration rules from the source to the target model and vice versa. Moreover, these coupled rules can be composed to create compound transformations at the model level that induce compound transformations at the instance level. We have implemented this technique in the HAEXCEL framework (available from the first author's web page: <http://www.di.uminho.pt/~jacome>): a set of HASKELL-based libraries and tools to manipulate spreadsheets. With this approach, spreadsheet evolution can be made type-safe, also when model changes are involved.

The rest of this paper is organized as follows. In Section 2 we discuss spreadsheet refactoring as our motivating example. In Section 3 we describe the framework to model and manipulate spreadsheets. Section 4 defines the rules to perform the evolution of spreadsheets. Section 5 discusses related work and Section 6 concludes the paper.

2 Motivating Example: Spreadsheet Refactoring

Suppose a researcher's yearly budget for travel and accommodation expenses is kept in the spreadsheet shown in Figure 1 taken from [11].

Note that throughout the years, cost and quantity are registered for three types of expenses: travel, hotel and local transportation. Formulas are used to calculate the total expense for each type in each year as well as the total expense in each year. Finally a grand total is calculated over all years, both per type of expense and overall.

At the end of 2010, this spreadsheet needs to be modified to accommodate 2011 data. A novice spreadsheet user would typically take four steps to perform the necessary task:

	A	B	C	D	E	F	G	H	I
1	Budget		Year			Year			
2			Year=2010			Year=2011			
3	Category	Name	Qty	Cost	Total	Qty	Cost	Total	Total
4	travel		2	200	400	2	450	900	1300
5	hotel		5	100	500	8	80	640	1140
6	local travel		4	20	80	2	35	70	150
7	Total				980			1610	2590

Fig. 1. Budget spreadsheet instance

insert three new columns; copy all the labels; copy all the formulas (at least two); update all the necessary formulas in the last column. A more advanced user would shortcut these steps by copy-inserting the 3-column block of 2010 and changing the label “2010” to “2011” in the copied block. If the insertion is done behind the last year, the range of the multi-year totals columns must be extended to include the new year. If the insertion is done in between the last and one-but-last year, the spreadsheet system automatically extends the formulas for the multi-year totals. Apart from these two strategies, a mixed strategy may be employed. In any case, a conceptually unitary modification (*add year*) needs to be executed by an error-prone combination of steps.

Erwig *et al.* have introduced *ClassSheets* as models of spreadsheets that allow spreadsheet modifications to be performed at the right conceptual level. For example, the ClassSheet in Figure 2 provides a model of our budget spreadsheet.

	A	B	D	E	F	...	G
1	Budget	Year					
2		year=2010					
3	Category	Name	Qty	Cost	Total	Total	
4	name="abc"	qty=0	cost=0	total=qty*cost		total=SUM(total)	
5	Total			total=SUM(total)		total=SUM(Year.total)	

Fig. 2. Budget spreadsheet model

In this model, the repetition of a block of columns for each year is captured by gray column labeled with the ellipsis. The horizontal repetition is marked in a analogous way. This makes it possible (i) to check whether the spreadsheet after modification still instantiates the same model, and (ii) to offer the user an unitary operation. Apart from (horizontal) block repetitions that support the extension with more years, this model features (vertical) row repetitions that support the extension with new expense types.

Unfortunately, situations may occur in which the model itself needs to be modified. For example, if the researcher needs to report expenses before and after tax, additional columns need to be inserted in the block of each year. Figure 3 shows the new spreadsheet as well as the new model that it instantiates.

Note that a modification of the year block in the model (inserting various columns) captures modifications to all repetitions of the block throughout the instance.

In this paper, we will demonstrate that modifications to spreadsheet models can be supported by an appropriate combinator language, and that these model modifications can be propagated automatically to the spreadsheets that instantiate the models. In case of the budget example, the model modification is captured by the following expression:

Cost	Tax tariff	After tax	Total
cost=0	tax tariff=0	after tax=cost+cost*tariff	total=qnty*cost
			total=SUM(total)

Cost	Tax tariff	After tax	Total
200	0,12	224	400
100	0,2	120	500
20	0,2	24	80
			980

(a) New budget model

(b) New budget instance

Fig. 3. New spreadsheet and the model that it instantiates

```
addTax = once (inside "Year" (before "Total" (
    insertCol "Tax Tariff" ▷ insertCol "After tax")))
```

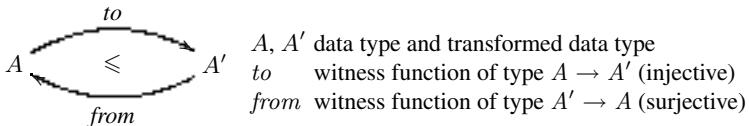
The actual column insertions are done by the innermost sequence of two *insertCol* steps. The *before* and *inside* combinators specify the location constraints of applying these steps. The *once* combinator traverses the spreadsheet model to search for a single location where these constraints are satisfied and the insertions can be performed.

Application of the *addTax* transformation to the initial model (Figure 2) will yield: firstly, the modified model (Figure 3a), secondly a spreadsheet migration function that can be applied to instances of the initial model (*e.g.* Figure 1) to produce instances of the modified model (*e.g.* Figure 3b), and thirdly an inverse spreadsheet migration function to backport instances of the modified model to instances of the initial model.

In the remainder of this paper, we will explain the machinery required for this type of coupled transformation of spreadsheet instances and models. As models, we will use a variation on ClassSheets where references are modeled by projection functions. Model transformations propagate references by composing instance-level transformations with these projection functions.

3 A Framework for Evolution of Spreadsheets in HASKELL

Data refinement theory provides an algebraic framework for calculating with data types and corresponding values [16,17,18]. It consists of type-level coupled with value-level transformations. The type-level transformations deal with the evolution of the model and the value-level transformations deal with the instances of the model (*e.g.* values). Figure 4 depicts the general scenario of a transformation in this framework.

**Fig. 4.** Coupled transformation of data type A into data type A'

Each transformation is coupled with witness functions *to* and *from*, which are responsible for converting values of type A into type A' and back.

The 2LT framework is an HASKELL implementation of this theory [3,4,5,6]. It provides the basic combinators to define and compose transformations for data types and witness functions. Since 2LT is statically typed, transformations are guaranteed to be type-safe ensuring consistency of data types and data instances.

3.1 ClassSheets and Spreadsheets in HASKELL

The 2LT was originally designed to work with algebraic data types. However, this representation is not expressive enough to represent ClassSheet specifications or their spreadsheet instances. To overcome this issue, we extended the 2LT representation so it could support ClassSheet models, by introducing the following *Generalized Algebraic Data Type*¹ (GADT) [12,20]:

```

data Type a where
  ...
  Value :: Value → Type Value                                -- plain value
  RefCell :: Type RefCell                                     -- references
  Ref      :: Type b → PF (a → RefCell) → PF (a → b) → Type a → Type a
  Formula :: Formula → Type Formula                         -- reference cell
  LabelB :: String → Type LabelB                           -- formulas
  · = · :: Type a → Type b → Type (a, b)                  -- block label
  · ∙ · :: Type a → Type b → Type (a, b)                  -- attributes
  · ∙ · :: Type a → Type b → Type (a, b)                  -- block horizontal composition
  · ^ · :: Type a → Type b → Type (a, b)                  -- block vertical composition
  EmptyB :: Type EmptyB                                    -- empty block
  · ∙ :: String → Type HorH                             -- horizontal class label
  | · :: String → Type VerV                            -- vertical class label
  | ∙ :: String → Type Square                          -- square class label
  LabRel :: String → Type LabS                           -- relation class
  · ∙ · :: Type a → Type b → Type (a, b)                  -- labeled class
  · ∙ (·)↓ :: Type a → Type b → Type (a, [b])          -- labeled expandable class
  · ^ · :: Type a → Type b → Type (a, b)                  -- class vertical composition
  SheetC :: Type a → Type (SheetC a)                      -- sheet class
  · → :: Type a → Type [a]                               -- sheet expandable class
  · ∙ · :: Type a → Type b → Type (a, b)                  -- sheet horizontal composition
  EmptyS :: Type EmptyS                                    -- empty sheet

```

The comments should clarify what the constructors represent. The values of type *Type a* are representations of type *a*. For example, if *t* is of type *Type Value*, then *t* represents the type *Value*. The following types are needed to construct values of type *Type a*:

```

data EmptyBlock                                         -- empty block
data EmptySheet                                         -- empty sheet
type LabelB = String                                  -- label
data RefCell = RefCell1                                -- referenced cell
type LabS = String                                   -- square label
type HorH = String                                   -- horizontal label
type VerV = String                                   -- vertical label
data SheetC a = SheetCC a                            -- sheet class
data SheetCE a = SheetCEC a                          -- expandable sheet class
data Value = VInt Int | VString String | VBool Bool | VDouble Double -- values
data Formula1 = FValue Value | FRef | FFormula String [Formula1] -- formula

```

¹ “It allows to assign more precise types to data constructors by restricting the variables of the datatype in the constructors’ result types.”

Once more, the comments should clarify what each type represents.

To explain this representation we will use as an example a reduced version of the budget model presented in Figure 1. For this reduced model only three columns were defined: *quantity*, *cost per unit* and *total cost* (product of *quantity* by *cost per unit*).

```
purchase =
| Price_List : Quantity + Price + Total ^
| PriceList : (quantity = 0 + price = 0 + total = FFormula × [FRef, FRef])↓
```

This ClassSheet specifies a class called *Price_List* composed by two parts vertically composed as indicated by the \wedge operator. The first part is defined in the first row and defines the labels for three columns: *Quantity*, *Price* and *Total*. The second row defines the rest of the class containing the definition of the three columns. The first two columns have as default value 0 and the third is defined by a formula (explained latter on). Note that this part is vertical expandable, that is, it can be vertically repeated. In a spreadsheet instance this corresponds to the possibility of adding new rows. Figure 5 represents a spreadsheet instance of this model.

	A	B	C
1	Quantity	Price	Total
2	2	1500	=A2*B2
3	5	45	=A3*B3

Fig. 5. Spreadsheet instance of the *purchase* ClassSheet

Note that in the definition of *Type a* the constructors combining parts of the spreadsheet (e.g. sheets) return a pair. Thus, a spreadsheet instance is written as nested pairs of values. The spreadsheet illustrated in Figure 5 is encoded in HASKELL as follows:

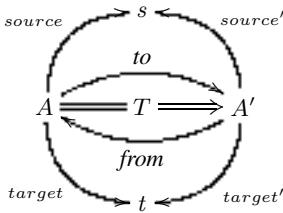
```
((Quantity, (Price, Total)),  
[(2, (1500, FormulaFF × [FRef, FRef])),  
(5, (45, FormulaFF × [FRef, FRef])))])
```

The HASKELL type checker statically ensures that the pairs are well formed and are constructed in the correct order.

3.2 Specifying Formulas

Having defined a GADT to represent ClassSheet models, we need now a mechanism to define spreadsheet formulas. The safer way to specify formulas is making them strongly typed. Figure 6 depicts the scenario of a transformation with references. A reference from a cell *s* to the a cell *t* is defined using a pair of projections, *source* and *target*. These projections are statically-typed functions traversing the data type *A* to identify the cell defining the reference (*s*), and the cell to which the reference is pointing to (*t*). In this approach, not only the references are statically typed, but also always guaranteed to exist, that is, one can not create a reference from/to a cell that does not exist.

The projections defining the reference and the referenced type, in the transformed type *A'*, are obtained by post-composing the projections with the witness function *from*.



source Projection over type A identifying the reference
 target Projection over type A identifying the referenced cell
 $\text{source}' = \text{source} \circ \text{from}$
 $\text{target}' = \text{target} \circ \text{from}$

Fig. 6. Coupled transformation of data type A into data type A' with references

When source' and target' are normalized they work on A' directly rather than via A . The formula specification, as previously shown, is specified directly in the GADT. However, the references are defined separately by defining projections over the data type. This is required to allow any reference to access any part of the GADT.

Using the spreadsheet illustrated in Figure 5, an instance of a reference from the formula *total* to *price* is defined as follows (remember that the second argument of *Ref* is the source (reference cell) and that the third is the target (referenced cell)):

```

purchaseWithReference =
  Ref Int (fhead ∘ head ∘ (π₂ ∘ π₂)*) ∘ π₂) (head ∘ (π₁ ∘ π₂)*) ∘ π₂) purchase

```

The *source* function refers to the first *FRef* in the HASKELL encoding shown after Figure 5. The *target* projection defines the cell it is pointing to, that is, it defines a reference to the value 1500 in column *Price*. Since the use of GADTs requires the definition of models combining elements in a pairwise fashion, it is necessary to descend into the structure using π_1 and π_2 . The operator \cdot^* applies a function to all the element of a list and *fhead* gets the first reference in a list of references.

Note that our reference type has enough information about the cells and so we do not need value-level functions, that is, we do not need to specify the projection functions themselves, just their types. In the cases we reference a list of values, for example, constructed by the class expandable operator, we need be specific about the element within the list we are referencing. For these cases, we use the type-level constructors *head* (first element of a list) and *tail* (all but first) to get the intended value in the list.

3.3 Rewriting Systems

At this point we are now able to represent ClassSheet models, including formulas. In this section we discuss the definition of the witness functions *from* and *to*. Once again we rely on the definition of a GADT:

```

data PF a where
  id      :: PF (a → a)                                -- identity function
  π₁     :: PF ((a, b) → a)                            -- left projection of a pair
  π₂     :: PF ((a, b) → b)                            -- right projection of a pair
  pnt   :: a → PF (One → a)                           -- constant
  · △ ·  :: PF (a → b) → PF (a → c) → PF (a → (b, c)) -- split of functions
  · × ·  :: PF (a → b) → PF (c → d) → PF ((a, c) → (b, d)) -- product of functions

```

```

 $\cdot \circ \cdot$  :: Type  $b \rightarrow PF(b \rightarrow c) \rightarrow PF(a \rightarrow b) \rightarrow PF(a \rightarrow c)$       -- composing func.
 $\cdot^*$        ::  $PF(a \rightarrow b) \rightarrow PF([a] \rightarrow [b])$                                 -- map of functions
head        ::  $PF([a] \rightarrow a)$                                               -- head of a list
tail        ::  $PF([a] \rightarrow [a])$                                          -- tail of a list
fhead       ::  $PF(Formula1 \rightarrow RefCell)$                          -- head of the arguments of a formula
ftail       ::  $PF(Formula1 \rightarrow Formula1)$                          -- tail of the arguments of a formula

```

This GADT represents the types of the functions used in the transformations. For example, π_1 represents the type of the function that projects the first part of a pair. The comments should clarify which function each constructor represents. Given these representations of types and functions, we can turn to the encoding of refinements. Each refinement is encoded as a two-level rewriting rule:

```

type Rule =  $\forall a . Type a \rightarrow Maybe(View(Type a))$ 
data View a where View :: Rep a b  $\rightarrow$  Type  $b \rightarrow View(Type a)$ 
data Rep a b = Rep {to =  $PF(a \rightarrow b)$ , from =  $PF(b \rightarrow a)$ }

```

Although the refinement is from a type a to a type b , this can not be directly encoded since the type b is only known when the transformation completes, so the type b is represented as a *view* of the type a . A *view* expresses that a type a can be represented as a type b , denoted as $Rep a b$, if there are functions $to :: a \rightarrow b$ and $from :: b \rightarrow a$ that allow data conversion between one and the other. The following code implements a rule to transform a list into a map (represented by $\cdot \multimap \cdot$):

```

listmap :: Rule
listmap ([a]) = Just (View (Rep {to = seq2index, from = tolist})) (Int  $\multimap$  a)
listmap _ = mzero

```

The witness functions have the following signature (their code here is not important):

```
tolist :: (Int  $\multimap$  a)  $\rightarrow$  [a]           seq2index :: [a]  $\rightarrow$  Int  $\multimap$  a
```

This rule receives the type of a list of a , $[a]$, and returns a view over the type map of integers to a , $Int \multimap a$. The witness functions are returned in the representation *Rep*. If other argument than a list is received, then the rule fails returning *mzero*. All the rules contemplate this case and so we will not show it in the definition of other rules.

Given this encoding of individual rewrite rules, a complete rewrite system can be constructed via the following constructors:

```

nop :: Rule          -- identity
▷ :: Rule  $\rightarrow$  Rule  $\rightarrow$  Rule    -- sequential composition
∅ :: Rule  $\rightarrow$  Rule  $\rightarrow$  Rule    -- left-biased choice
many :: Rule  $\rightarrow$  Rule          -- repetition
once :: Rule  $\rightarrow$  Rule          -- arbitrary depth rule application

```

Details on the implementation of these combinators can be found elsewhere [4].

4 Spreadsheets Evolution

In this section we define rules to perform spreadsheet evolution. These rules can be divided in three main categories: *Combinators*, used as helper rules, *Semantic* rules, intended to change the model itself (*e.g.* add a new column), and *Layout* rules, designed to change the visual arrangement of the spreadsheet (*e.g.* swap two columns).

4.1 Combinators

The other types of rules are defined to work on a specific part of the data type. The combinators defined next are then used to apply those rules in the desired places.

Pull Up All the References: To avoid having references in different levels of the models, all the rules pull all the references to the topmost level of the model. To pull a reference is a particular place we use the following rule (we show just its first case):

```
pullUpRef :: Rule
pullUpRef ((Ref tb fRef tRef ta) : b2) = do
    return (View idrep (Ref tb (fRef ∘ π1) (tRef ∘ π1) (ta : b2)))
```

The representation *idrep* has the *id* function in both directions. If part of the model (in this case the left part of a horizontal composition) of a given type has a reference, it is pulled to the top level. This is achieved by composing the existing projections with the necessary functions, in this case π_1 . This rule has two cases (left and right hand side) for each binary constructor (*e.g.* horizontal/vertical composition).

To pull up all the references in all levels of a model we use the rule $pullUpAllRefs = many$ (*once* $pullUpRef$). The *once* operator applies the $pullUpRef$ rule somewhere in the type and the *many* ensures that this is applied everywhere in the whole model.

Apply After and Friends: The combinator *after* finds the correct place to apply the argument rule (second argument) by comparing the given string (first argument) with the existing labels in the model. When it finds the intended place, it applies the rule to it. This works because our rules always do their task on the right-hand side of a type.

```
after :: String → Rule → Rule
after label r (label' : a) | label ≡ label' = do
    View s l' ← r label'
    return (View (Rep {to = to s × id, from = from s × id}) (l' : a))
```

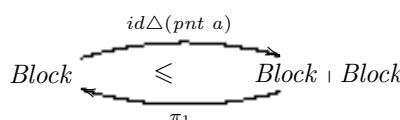
Note that this definition is only part of the complete version since it only contemplates the case for horizontal composition of blocks $(\cdot \mid \cdot)$.

Other combinators were also developed, namely, *before*, *bellow*, *above*, *inside* and *at*. Their implementations are not shown since they are similar to the *after* combinator.

4.2 Semantic Rules

In this section we present rules that change the semantics of the model, for example, adding columns.

Insert a Block: One of the most fundamental rules is the insertion of a new block into a spreadsheet, formally defined as following:



This diagram means that a horizontal composition of two blocks refines a block when witnessed by two functions, *to* and *from*. The *to* function, $\text{id}\Delta(pnt\ a)$, is a split: it injects the existing block in the first part of the result without modifications (*id*) and injects the given block instance *a* into the second part of the result. The *from* function is π_1 since it is the one that allows the recovery of the existent block. The HASKELL version of the rule is presented next.

```
insertBlock :: Type a → a → Rule
insertBlock ta a tx | isBlock ta ∧ isBlock tx = do
  let rep = Rep { to = (idΔ(pnt a)), from =  $\pi_1$  }
  View s t ← pullUpAllRefs (tx + ta)
  return (View (comprep rep s) t)
```

The function *comprep* composes two representations. This rule receives the type of the new block *ta*, its default instance *a*, and returns a *Rule*. The returned rule is itself a function that receives the block to modify *tx* and returns a view of the new type. The first step is to verify if the given types are block using the function *isBlock*. The second step is to create the representation *rep* with the witness functions given in the above diagram. Then the references are pulled up in result type *tx* + *ta*. This returns a new representation *s* and a new type *t* (in fact, the type is the same *t* = *tx* + *ta*). The result view has as representation the composition of the two previous representations, *rep* and *s*, and the corresponding type *t*.

Rules to insert classes and sheets were also defined, but since these rules are similar to the rule for inserting blocks, we omit them for brevity.

Insert a Column: To insert a column in a spreadsheet, that is, a cell with a label *lbl* and the cell bellow with a default value *df* and vertically expandable, we first need to create a new class representing it: *clas* =| *lbl* : *lbl* ^ (*lbl* = df^\downarrow). The label is used to create the default value (*lbl*, []). Note that, since we want to create an expandable class, the second part of the pair must be a list. The final step is to apply *insertSheet*:

```
insertCol :: String → VFormula → Rule
insertCol l f@(FFormula name fs) tx | isSheet tx = do
  let clas =| lbl : lbl ^ (lbl =  $df^\downarrow$ )
  ((insertSheet clas (lbl, [])) ▷ pullUpAllRefs) tx
```

Note the use of the rule *pullUpAllRefs* as explained before. The case shown in the above definition is for a formula as default value and it is similar to the value case. The case with a reference is more interesting and is shown next:

```
insertCol l FRef tx | isSheet tx = do
  let clas =| lbl : Ref ⊥ ⊥ ⊥ (lbl ^ ((lbl = RefCell) $^\downarrow$ ))
  ((insertSheet clas (lbl, [])) ▷ pullUpAllRefs) tx
```

Recall that our references are always local, that is, they can only exist with the type they are associated with. So, it is not possible to insert a column that references a part of the existing spreadsheet. To overcome this, we first create the reference with undefined functions and auxiliary type (\perp) and then we set these values to the intended ones.

```
setFormula :: Type b → PF (a → RefCell) → PF (a → b) → Rule
setFormula tb fRef tRef (Ref — — t) = return (View idrep (Ref tb fRef tRef t))
```

This rule receives the auxiliary type (*Type b*), the two functions representing the reference projections and adds them to the type. A complete rule to insert a column with a reference is defined as follows:

```
insertFormula =
  (once (insertCol "After Tax" FRef)) ▷ (setFormula auxType fromRef toRef)
```

Following the original idea described in Section 2, we want to introduce a new column with the tax tariff. In this case, we want to insert a column in an existing block and thus our previous rule will not work. For these cases we write a new rule:

```
insertColIn :: String → VFormula → Rule
insertColIn l (FValue v) tx | isBlock tx = do
  let block = lbl^ (lbl = v)
  ((insertBlock block (lbl, v)) ▷ pullUpAllRefs) tx
```

This rule is similar to the previous one but it creates a block (not a class) and inserts it also after a block. The reasoning is analogous to the one in *insertCol*.

To add the two columns "Tax tariff" and "After tax" we can use the rule *insertColIn*, but applying it directly to our running example will fail since it expects a block and we have a spreadsheet. We can use the combinator *once* to achieve the desired result. This combinator tries to apply a given rule somewhere in a type, stopping after it succeeds once. Although this combinator already existed in the 2LT framework, we extended it to work for spreadsheet models. Assuming that the column "Tax tariff" was already inserted, we can run the following functions:

```
ghci>let formula = FFormula × [FRef, FRef]
ghci>once (after "Tax tariff" (once (insertColIn "After Tax" formula))) budget
...
("Cost" ∘ "Tax tariff" ∘ "After tax" ^ ("after tax" = formula) ∘ "Total") ^
("cost" = 0 ∘ "tax tarif" = 0 ∘ "total" = totalFormula)
...
```

Note that above result is not quite right. The block inserted is a vertical composition and is inserted in a horizontal composition. The correct would be to have its top and bottom part on the top and bottom part of the result, as defined below:

```
("Cost" ∘ "Tax tariff" ∘ "After tax" ∘ "Total") ^
("cost" = 0 ∘ "tax tarif" = 0 ∘ "after tax" = formula ∘ "total" =
 totalFormula)
```

To correct these cases, we designed a layout rule, *normalize*, explained in Section 4.3.

Make it Expandable: It is possible to make a block in a class expandable. For this, we created the rule *expandBlock*:

```
expandBlock :: String → Rule
expandBlock str (label : clas) | compLabel label str = do
  let rep = Rep {to = id × tolist, from = id × head}
  return (View rep (label : (clas)↓))
```

It receives the label of the class to make expandable and updates the class to allow repetition. The result type constructor is $\cdot : (\cdot)^{\downarrow}$; the *to* function wraps the existing

block into a list, *toList*; and the *from* function takes the head of it, *head*. We developed a similar rule to make a class expandable. This corresponds to promote a class *c* to c^\rightarrow . We do not show its implementation here since it is quite similar to this one.

Split: It is quite common to move a column in a spreadsheet from one place to another. The rule *split* copies a column to another place and substitutes the original column values by references to the new column (similar to create a pointer). The rule to move part of the spreadsheet is presented in Section 4.3. The first step of *split* is to get the column that we want to copy:

```
getColumn :: String → Rule
getColumn h t (l^b1) | h ≡ l' = return (View idrep t)
```

If the corresponding label is found, the vertical composition is returned. Note that, as in other rules, this rule is intended to be applied using the combinator *once*. As we said, we aim to write local rules that can be used at any level using the developed combinators.

The rule creates in a second step a new a class containing the retrieved block:

```
do View s c' ← getBlock str c
  let nsh =| str : (c')^\downarrow
```

The last step is to transform the original column that was copied into references to the new column. The rule *makeReferences* :: *String* → *Rule* receives the label of the column that was copied (the same as the new column) and creates the references. We do not show the rest of the implementation because it is quite complex and will not help in the understanding of the paper.

Let us consider the following part of our example:

```
budget =
... ("Cost" | "Tax tariff" | "After tax" | "Total")^
  ("cost" = 0 | "tax tariff" = 0 | "after tax" = formula | "total" =
   totalFormula) ...
```

If we apply the *split* rule (with the help of *once*) to it we get the following new model:

```
ghci> once (split "Tax tariff") budget
...
("Cost" | "Tax tariff" | "After tax" | "Total")^
  ("cost" = 0 | "tax tariff" = 0 | RefCell | "total" = totalFormula)
  |
  ("Tax tariff" : ((("Tax tariff" ^ "tax tariff" = 0))^\downarrow))
```

4.3 Layout Rules

In this section we describe rules focused on the layout of spreadsheets, that is, rules that do not add/remove information to/from the model.

Change Orientation: The rule *toVertical* changes the orientation of a block from horizontal to vertical.

toVertical :: Rule

toVertical (a ∘ b) = return (View idrep (a ^ b))

Note that, since our value-level representation of these compositions are pairs, the *to* and the *from* functions are simply the identity function. The needed information is kept in the type-level with the different constructors. A rule to do the inverse was also designed but since it is quite similar to this one, we do not show it here.

Normalize Blocks: When applying some transformations, the resulting types may not have the correct shape. A common example is to have as result the following type:

$$\begin{array}{c} A \downarrow B \wedge C \downarrow D \\ E \downarrow F \end{array}$$

Most of the times, the correct result is the following:

$$\begin{array}{c} A \downarrow B \downarrow D \\ E \downarrow C \downarrow F \end{array}$$

The rule *normalize* tries to match these cases and correct them. The types are the ones presented above and the witness functions are combinations of π_1 and π_2 .

normalize1 :: Rule

normalize1 (a ∘ b ^ c ∘ d ^ e ∘ f) =

```
let tof = id ×  $\pi_1 \times id \circ \pi_1 \Delta \pi_1 \circ \pi_2 \Delta \pi_2 \circ \pi_1 \circ \pi_2 \times \pi_2$ 
    fromf =  $\pi_1 \circ \pi_1 \Delta \pi_1 \circ \pi_2 \times \pi_1 \circ \pi_2 \Delta \pi_2 \circ \pi_2 \circ \pi_1 \Delta id \times \pi_2 \circ \pi_2 \circ \pi_2$ 
return (View (Rep {to = tof, from = fromf}) (a ∘ b ∘ d ^ e ∘ c ∘ f))
```

Although the migration functions seem complex, they just rearrange the order of the pair so they have the correct order.

Shift: It is quite common to move parts of the spreadsheet across it. We designed a rule to shift parts of the spreadsheet in the four possible directions. We show here part of the *shiftRight* rule, which, as suggested by its name, shifts a piece of the spreadsheet to the right. In this case, a block is moved and an empty block is left in its place.

shiftRight :: Type a → Rule

shiftRight ta b1 | isBlock b1 = do

Eq ← teq ta b1

let rep = Rep {to = pnt (⊥ :: EmptyBlock) Δ id, from = π2}

return (View rep (EmptyBlock ∘ b1))

The function *teq* verifies if two types are equal. This rule receives a type and a block, but we can easily write a wrapper function to receive a label in the same style of *insertCol*.

Another interesting case of this rules occurs when the user tries to move a block (or a sheet) that has a reference.

shiftRight ta (Ref tb frRef toRef b1) | isBlock b1 = do

Eq ← teq ta b1

let rep = Rep {to = pnt (⊥ :: EmptyBlock) Δ id, from = π2}

return (View rep (Ref tb (frRef ∘ π2) (toRef ∘ π2) (EmptyBlock ∘ b1)))

As we can see in the above code, the existing reference projections must be composed with the selector π_2 to allow to retrieve the existing block *b1*. Only after this it is possible to apply the defined selection reference functions.

Move Blocks: A more complex task is to move a part of the spreadsheet to another place. We present next a rule to move a block.

```
moveBlock :: String → Rule
moveBlock str c = do View s c' ← getBlock str c
                     let nsh =| str : c'
                     View r sh ← once (removeRedundant str) (c ∪ nsh)
                     return (View (comprep s r) sh)
```

After getting the intended block and creating a new class with it, we need to remove the old block using *removeRedundant*.

```
removeRedundant :: String → Rule
removeRedundant s (s') | s ≡ s' = return (View rep EmptyBlock)
where rep = Rep {to = pnt (⊥ :: EmptyBlock), from = pnt s'}
```

This rule will remove the block with the given label leaving an empty block in its place.

5 Related Work

Ko *et al.* [13] summarize and classify the research challenges of the end user software engineering area. These include requirements gathering, design, specification, reuse, testing and debugging. However, besides the importance of Lehman's laws of software evolution [14], very little is stated with respect to spreadsheet evolution. Spreadsheets evolution poses challenges not only in the evolution of the underlying model, but also in migration of the spreadsheet values and used formulae. Nevertheless, many of the underlying transformations used for spreadsheet transformations are shared with works for spreadsheet generation and other program transformation techniques.

Engels *et al.* [15] propose a first attempt to solve the problem of spreadsheet evolution. ClassSheets are used to specify the spreadsheet model and transformation rules are defined to enable model evolution. These model transformations are propagated to the model instances (spreadsheets) through a second set of rules which updates the spreadsheet values. They present a set of rules and a prototype tool to support these changes. In this paper we present a more advanced way to evolve spreadsheets models and instances in a different way: first, we use strategic programming with two-level coupled transformation. This enables type-safe transformations, offering guarantee that in any step semantics are preserved. Also, the use of 2LT not only gives for free the data migration but also it allows back portability, that is, it allows the migration of data from the new model back to the old model.

Vermolen and Visser [23] proposed a different approach for coupled evolution of data model and data. From a data model definition, they generate a domain specific language (DSL) which supports the basic transformations and allows data model and data evolution. The interpreter for the DSL is automatically generated making this approach operational. This approach could also be used for spreadsheet evolution. However, there are a few important differences. While their approach is tailored for forward evolution, our approach supports reverse engineering, that is, it supports automatic transformation and migration from the new model to the old model.

6 Conclusions

In this paper, we have presented an approach for disciplined model-driven evolution of spreadsheets. The approach takes as starting point the observation that spreadsheets can be seen as instances of a spreadsheet model capturing the business logic of the spreadsheet. We have extended the calculus for coupled transformations of the 2LT platform to this spreadsheet model. An important novel aspect of this extension is the treatment of references. In particular, we have made the following contributions:

- We have provided a model of spreadsheets in the form of a GADT with embedded point-free function representations. This model is reminiscent of the ClassSheet.
- We have defined a coupled transformation system in which transformations at the level of spreadsheet models are coupled with corresponding transformations at the level of spreadsheet data/instances. This system combines strategy combinators known from strategic programming with spreadsheet-specific transformation rules.
- We have illustrated our approach with a number of specific spreadsheet refactorings to perform the evolution of spreadsheets.

The rules here presented are implemented in the HAEXCEL framework consisting of a set of libraries providing functionality to load (from different formats), transform, infer spreadsheet models (e.g. ClassSheet), and, now, perform the co-evolution of such models their (spreadsheet) instances. HAEXCEL includes an add-on for OpenOffice. Currently, we are integrating the rules presented in this paper, in a spreadsheet programming environment where end users can interact both with the model and the data [9].

References

1. Abraham, R., Erwig, M.: Inferring templates from spreadsheets. In: Proc. of the 28th Int. Conference on Software Engineering, pp. 182–191. ACM, New York (2006)
2. Abraham, R., Erwig, M., Kollmansberger, S., Seifert, E.: Visual specifications of correct spreadsheets. In: Proc. of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 189–196. IEEE Computer Society, Washington, DC, USA (2005)
3. Alves, T., Silva, P., Visser, J.: Constraint-aware Schema Transformation. In: The Ninth International Workshop on Rule-Based Programming (2008)
4. Cunha, A., Oliveira, J.N., Visser, J.: Type-safe two-level data transformation. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 284–299. Springer, Heidelberg (2006)
5. Cunha, A., Visser, J.: Strongly typed rewriting for coupled software transformation. Electronic Notes on Theoretical Computer Science 174, 17–34 (2007)
6. Cunha, A., Visser, J.: Transformation of structure-shy programs: Applied to XPath queries and strategic functions. In: Ramalingam, G., Visser, E. (eds.) Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation 2007, Nice, France, January 15–16, pp. 11–20. ACM, New York (2007)
7. Cunha, J., Beckwith, L., Fernandes, J.P., Saraiva, J.: An empirical study on the influence of different spreadsheet models on end-users performance. Tech. Rep. DI-CCTC-10-10, CCTC, Departamento de Informática, Universidade do Minho (2010)

8. Cunha, J., Erwig, M., Saraiva, J.: Automatically inferring classsheet models from spreadsheets. In: Proc. of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 93–100. IEEE Computer Society, Washington, DC, USA (2010)
9. Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J.: Embedding spreadsheet models in spreadsheet systems (2011) (submitted for publication)
10. Cunha, J., Saraiva, J., Visser, J.: Discovery-based edit assistance for spreadsheets. In: Proc. of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 233–237. IEEE Computer Society, Washington, DC, USA (2009)
11. Engels, G., Erwig, M.: ClassSheets: Automatic generation of spreadsheet applications from object-oriented specifications. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 124–133. ACM, New York (2005)
12. Hinze, R., Löh, A., Oliveira, B.: “Scrap Your Boilerplate” Reloaded. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, pp. 13–29. Springer, Heidelberg (2006)
13. Ko, A., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrence, J., Lieberman, H., Myers, B., Rosson, M., Rothermel, G., Shaw, M., Wiedenbeck, S.: The state of the art in end-user software engineering. *J. ACM Computing Surveys* (2009)
14. Lehman, M.M.: Laws of software evolution revisited. In: Montangero, C. (ed.) EWSPT 1996. LNCS, vol. 1149, pp. 108–124. Springer, Heidelberg (1996)
15. Luckey, M., Erwig, M., Engels, G.: Systematic evolution of typed (model-based) spreadsheet applications (submitted for publication)
16. Morgan, C., Gardiner, P.H.B.: Data refinement by calculation. *Acta Informatica* 27, 481–503 (1990)
17. Oliveira, J.N.: A reification calculus for model-oriented software specification. *Formal Aspects of Computing* 2(1), 1–23 (1990)
18. Oliveira, J.N.: Transforming data by calculation. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 134–195. Springer, Heidelberg (2008)
19. Panko, R.R.: Spreadsheet errors: What we know. What we think we can do. In: Proceedings of the Spreadsheet Risk Symposium, European Spreadsheet Risks Interest Group (July 2000)
20. Peyton Jones, S., Washburn, G., Weirich, S.: Wobbly types: type inference for generalised algebraic data types. Tech. Rep. MS-CIS-05-26, Univ. of Pennsylvania (July 2004)
21. Powell, S.G., Baker, K.R.: The Art of Modeling with Spreadsheets. John Wiley & Sons, Inc., New York (2003)
22. Rajalingham, K., Chadwick, D., Knight, B.: Classification of spreadsheet errors. In: European Spreadsheet Risks Interest Group, EuSpRIG (2001)
23. Vermolen, S.D., Visser, E.: Heterogeneous coupled evolution of software languages. In: Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 630–644. Springer, Heidelberg (2008)

A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications

Hartmut Ehrig¹, Claudia Ermel¹, and Gabriele Taentzer²

¹ Technische Universität Berlin, Germany

`claudia.ermel@tu-berlin.de, ehrig@cs.tu-berlin.de`

² Philipps-Universität Marburg, Germany

`taentzer@informatik.uni-marburg.de`

Abstract. In model-driven engineering, models are primary artifacts and can evolve heavily during their life cycle. Hence, versioning of models is a key technique which has to be offered by an integrated development environment for model-driven engineering. In contrast to text-based versioning systems, our approach takes abstract syntax structures in model states and operational features into account. Considering the abstract syntax of models as graphs, we define a model revision by a span $G \leftarrow D \rightarrow H$, called graph modification, where G and H are the old and new versions, respectively, and D the common subgraph that remains unchanged. Based on notions of behavioural equivalence and parallel independence of graph modifications, we are able to show a Local-Church-Rosser Theorem for graph modifications. The main goal of the paper is to handle conflicts of graph modifications which may occur in the case of parallel dependent graph modifications. The main result is a general merge construction for graph modifications that resolves all conflicts simultaneously in the sense that for delete-insert conflicts insertion has priority over deletion.

Keywords: graph modification, graph transformation, model versioning, conflict resolution.

1 Introduction

Visual models are primary artifacts in model-driven engineering. Like source code, models may heavily evolve during their life cycle and should put under version control to allow for concurrent modifications of one and the same model by multiple modelers at the same time. When concurrent modifications are allowed, contradicting and inconsistent changes might occur leading to versioning conflicts. Traditional version control systems for code usually work on file-level and perform conflict detection by line-oriented text comparison. When applied to the textual serialization of visual models, the result is unsatisfactory because the information stemming from abstract syntax structures might be destroyed and associated syntactic and semantic information might get lost.

Since the abstract syntax of visual models can be well described by graphs, we consider graph modifications to reason about model evolution. Graph modifications formalize the differences of two graphs before and after a change as a span of injective graph morphisms $G \leftarrow D \rightarrow H$ where D is the unchanged part, and we assume wlog. that $D \rightarrow G$ and $D \rightarrow H$ are inclusions. An approach to conflict detection based on graph modifications is described in [10]. We distinguish *operation-based* conflicts where deletion actions are in conflict with insertion actions and *state-based* conflicts where the tentative merge result of two graph modifications is not well-formed wrt. a set of language-specific constraints.

In this paper, we enhance the concepts of [10] by the resolution of operation-based conflicts of graph modifications. First of all, we define behavioural equivalence and parallel independence of graph modifications based on pushout constructions in analogy to algebraic graph transformations [3] and show a Local Church-Rosser Theorem for parallel independent graph modifications. Then we present a merge construction for conflict-free graph modifications and show that the merged graph modification is behavioural equivalent to the parallel composition of the given graph modifications.

The main new idea of this paper is a general merge construction for graph modifications which coincides with the conflict-free merge construction if the graph modifications are parallel independent. Our general merge construction can be applied to conflicting graph modifications in particular. We establish a precise relationship between the behaviour of the given modifications and the merged modification concerning deletion, preservation and creation of edges and nodes. In our main result, we show in which way different conflicts of the given graph modifications are resolved by the merge construction which gives insertion priority over deletion in case of delete-insert conflicts. Note, however, that in general the merge construction has to be processed further by hand, if other choices of conflict resolution are preferred for specific cases. Our running example is a model versioning scenario for statecharts where all conflicts are resolved by the general merge construction.

Structure of the paper: In Section 2, we present the basic concepts of algebraic graph modifications, including behavioural equivalence and a Local Church-Rosser Theorem. A general merge construction is presented and analysed in Section 3, where also the main result concerning conflict resolution is given¹. Related work is discussed in Section 4, and a conclusion including directions for future work is given in Section 5.

2 Graph Modifications: Independence and Behavioural Equivalence

Graph modifications formalize the differences of two graphs before and after a change as a span of injective graph morphisms $G \leftarrow D \rightarrow H$ where D is

¹ The long version of the paper containing full proofs is published as technical report [4].

the unchanged part. This formalization suits well to model differencing where identities of model elements are preserved for each element preserved. We recall the definition of graph modifications from [10] here:

Definition 1 (Graph modification). *Given two graphs G and H , a graph modification $G \xrightarrow{D} H$ is a span of injective morphisms $G \xleftarrow{g} D \xrightarrow{h} H$.*

Graph D characterizes an *intermediate graph* where all deletion actions have been performed but nothing has been added yet. Wlog. we can assume that g and h are inclusions, i.e. that D is a subgraph of G and of H . G is called *original graph* and H is called *changed or result graph*.

Example 1 (Graph modifications). Consider the following model versioning scenario for statecharts. The abstract syntax of the statechart in Figure 1 (a) is defined by the typed, attributed graph in Figure 1 (b). The node type is given in the top compartment of a node. The name of each node of type State is written in the attribute compartment below the type name. We model hierarchical statecharts by using containment edges. For instance, in Figure 1 (b), there are containment edges from superstate S_0 to its substates S_1 and S_2 ². Note that for simplicity of the presentation we abstract from transition events, guards and actions, as well as from other statechart features, but our technique can also be applied to general statecharts. Furthermore, from now on we use a compact notation of the abstract syntax of statecharts, where we draw states as nodes (rounded rectangles with their names as node ids) and transitions as directed arcs between state nodes. The compact notation of the statechart in Figure 1 (a) is shown in Figure 1 (c).

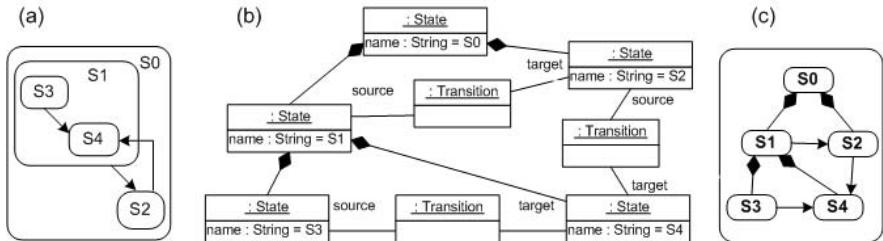


Fig. 1. Sample statechart: concrete syntax (a), abstract syntax graph (b), and compact notation (c)

In our model versioning scenario, two users check out the statechart shown in Figure 1 and change it in two different ways. User 1 performs a refactoring operation on it. She moves state S_3 up in the state hierarchy (cf. Figure 2). User 2 deletes state S_3 together with its adjacent transition to state S_4 .

Obviously, conflicts occur when these users try to check in their changes: state S_3 is deleted by user 2 but is moved to another container by user 1.

² In contrast to UML state machines, we distinguish edges that present containment links by composition decorators.

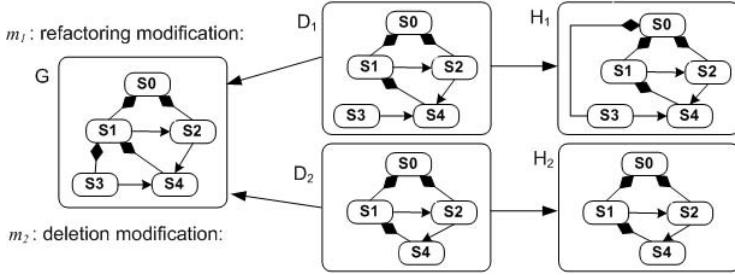


Fig. 2. Graph modifications \$m_1\$ (refactoring) and \$m_2\$ (deletion)

In this section, we study relations between different graph modifications based on category theory. Due to this general formal setting, we can use different kinds of graphs like labelled, typed or typed attributed graphs (see [3] for more details). At first, we consider the sequential and parallel composition of two graph modifications. Our intention is that graph modifications are closed under composition. Given the sequential composition of two graph modifications $G \xrightarrow{D_1} H_1$ and $H_1 \xrightarrow{D_2} H_2$, the resulting modification obviously has G as original and H_2 as changed graph. But how does their intermediate graph D should look like? The idea is to construct D as intersection graph of D_1 and D_2 embedded in H_1 . This is exactly realized by a pullback construction. The parallel composition of two graph modifications means to perform both of them independently of each other by componentwise disjoint union. This corresponds to coproduct constructions $G_1 + G_2$, $D_1 + D_2$ and $H_1 + H_2$ on original, intermediate and changed graphs.

Definition 2 (Composition of graph modifications). *Given two graph modifications $G \xrightarrow{D_1} H_1 = (G \leftarrow D_1 \rightarrow H_1)$ and $H_1 \xrightarrow{D_2} H_2 = (H_1 \leftarrow D_2 \rightarrow H_2)$, the sequential composition of $G \xrightarrow{D_1} H_1$ and $H_1 \xrightarrow{D_2} H_2$, written $(G \leftarrow D_1 \rightarrow H_1) * (H_1 \leftarrow D_2 \rightarrow H_2)$ is given by $G \xrightarrow{D} H_2 = (G \leftarrow D \rightarrow H_2)$ via the pullback construction $G \leftarrow D_1 \longrightarrow H_1 \leftarrow D_2 \longrightarrow H_2$.*

$$\begin{array}{ccccc} & & & & \\ & & & & \\ & \swarrow & \searrow & & \\ & (PB) & & & \\ & & & & \\ D & & & & \end{array}$$

The parallel composition of $G_1 \xrightarrow{D_1} H_1$ and $G_2 \xrightarrow{D_2} H_2$ is given by coproduct construction: $G_1 + G_2 \xrightarrow{D_1 + D_2} H_1 + H_2 = (G_1 + G_2 \leftarrow D_1 + D_2 \rightarrow H_1 + H_2)$.

The differences between the original and the intermediate graph as well as between the intermediate and the changed graph define the behaviour of a graph modification. The same behaviour can be observed in graphs with more or less context. Therefore, we define the behavioural equivalence of two graph modifications as follows: Starting with two modifications $m_i = (G_i \xrightarrow{D_i} H_i)$ ($i = 1, 2$), we look for a third graph modification $G \xrightarrow{D} H$ modeling the same changes with so little context that it can be embedded in m_1 and m_2 . A behaviourally equivalent embedding of graph modifications can be characterized best by two

pushouts as shown in Definition 3, since the construction of a pushout ensures that G_i are exactly the union graphs of G and D_i overlapping in D . Analogously, H_i are exactly the union graphs of H and D_i overlapping in D^3 .

Definition 3 (Behavioural Equivalence of Graph Modifications)

Two graph modifications $G_i \xrightarrow{D_i} H_i$ ($i = 1, 2$) are called behaviourally equivalent if there is a span $(G \leftarrow D \rightarrow H)$ and PO-span morphisms from $(G \leftarrow D \rightarrow H)$ to $(G_i \leftarrow D_i \rightarrow H_i)$, ($i = 1, 2$), i.e. we get four pushouts in the diagram to the right.

$$\begin{array}{ccccc} G_1 & \xleftarrow{\quad D_1 \quad} & H_1 \\ \uparrow \text{(PO)} & & \uparrow \text{(PO)} & & \uparrow \\ G & \xleftarrow{\quad D \quad} & H \\ \downarrow \text{(PO)} & & \downarrow \text{(PO)} & & \downarrow \\ G_2 & \xleftarrow{\quad D_2 \quad} & H_2 \end{array}$$

Example 2. Figure 3 shows two behaviourally equivalent graph modifications where the upper one is the refactoring modification m_1 from Figure 2. The span $(G \leftarrow D \rightarrow H)$ shows the same changes as in m_1 and m_2 , but in less context.

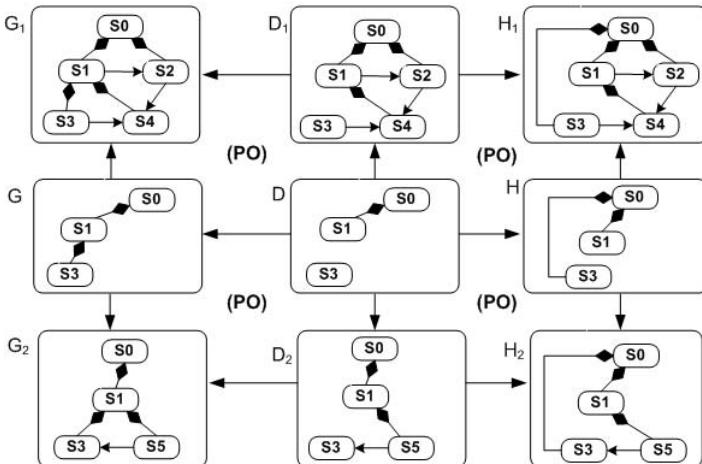


Fig. 3. Graph modifications m_1 and m_2 are behaviourally equivalent

We want to consider graph modifications to be parallel independent if they do not interfere with each other, i.e. one modification does not delete a graph element the other one needs to perform its changes. While nodes can always be added to a graph independent of its form, this is not true for edges. An edge can only be added if it has a source and a target node. Thus parallel independence means more concretely that one modification does not delete a node that is

³ In the framework of algebraic graph transformations [3], we may also consider graph modification $G \xrightarrow{D} H$ as a graph rule r which is applied to two different graphs G_1 and G_2 . Since the same rule is applied, graph transformations $G_i \xrightarrow{r} H_i$ ($i = 1, 2$) would be behaviourally equivalent.

supposed to be the source or target node of an edge to be added by the other modification. Moreover, both graph modifications could delete the same graph elements. It is debatable whether the common deletion of elements can still be considered as parallel independent or not. Since we consider parallel independent modifications to be performable in any order, common deletions are not allowed. Once modification m_1 has deleted a graph element, it cannot be deleted again by modification m_2 ⁴.

This kind of parallel independence is characterized by Definition 4 as follows: At first, we compute the intersection D of D_1 and D_2 in G by constructing a pullback. Since common deletions are not allowed, there has to be at least one modification for each graph element which preserves it. Thus, D_1 glued with D_2 via D has to lead to G , which corresponds to pushout (1). Next, considering D included in D_1 included in H_1 we look for some kind of graph difference. We want to identify those graph elements of H_1 that are not already in D_1 and have to be added by D_3 such that both overlap in D , i.e. H_1 becomes the pushout object of $D \rightarrow D_1$ and $D \rightarrow D_3$. In this case, D_3 with $D \rightarrow D_3 \rightarrow H_1$ is called pushout complement of $D \rightarrow D_1 \rightarrow H_1$ (see [3] for pushout and pushout complement constructions). Analogously, D_4 is the difference of H_2 and D_2 modulo D . Finally, both differences D_3 and D_4 are glued via D resulting in H . According to Proposition 1, both modifications may occur in any order, such that Definition 4 reflects precisely our intention of parallel independence.

Definition 4 (Parallel Independence of Graph Modifications)

Two graph modifications $(G \leftarrow D_i \rightarrow H_i)$, $(i = 1, 2)$ are called parallel independent if we have the four pushouts in the diagram to the right, where (1) can be constructed as pullback, (2) and (3) as pushout-complements and (4) as pushout.

$$\begin{array}{ccccc} G & \xleftarrow{\quad D_1 \quad} & H_1 \\ \uparrow & (1) & \uparrow & (2) & \uparrow \\ D_2 & \xleftarrow{\quad D \quad} & D_3 & & \\ \downarrow & (3) & \downarrow & (4) & \downarrow \\ H_2 & \xleftarrow{\quad D_4 \quad} & H & & \end{array}$$

Proposition 1 (Local Church-Rosser for Graph Modifications).

Given parallel independent graph modifications $G \xrightarrow{D_i} H_i$ ($i = 1, 2$). Then, there exists H and graph modifications $H_1 \xrightarrow{D_3} H$, $H_2 \xrightarrow{D_4} H$ which are behaviourally equivalent to $G \xrightarrow{D_2} H_2$, $G \xrightarrow{D_1} H_1$, respectively.

$$\begin{array}{ccc} G & \xrightarrow{\quad D_1 \quad} & H_1 \\ D_2 \parallel & & \parallel D_3 \\ H_2 & \xrightarrow{\quad D_4 \quad} & H \end{array}$$

Proof. Given parallel independent graph modifications $G \xrightarrow{D_i} H_i$ ($i = 1, 2$), we have pushouts (1) – (4) by Definition 4 leading to $H_1 \xrightarrow{D_3} H$ and $H_2 \xrightarrow{D_4} H$ with behavioural equivalence according to Definition 3. \square

Example 3. Figure 4 shows two parallel independent graph modifications where $m_1 = (G \leftarrow D_1 \rightarrow H_1)$ is the refactoring modification from Figure 2, and $m_2 = (G \leftarrow D_2 \rightarrow H_2)$ deletes the transition from S2 to S4 and adds a new state named S5. We have four pushouts, thus both graph modifications may be performed in any order, yielding in both cases the same result H .

⁴ However, we will see that modifications with common deletions still can be merged.

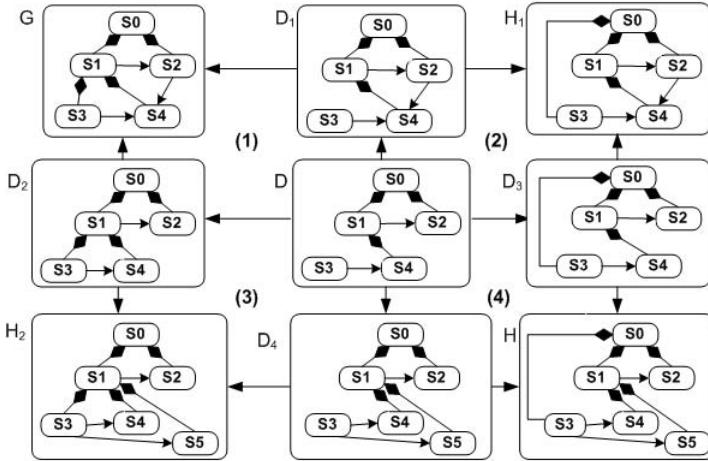


Fig. 4. Parallel independent graph modifications

In the case that two graph modifications are parallel independent, they are called *conflict-free* and can be merged to one *merged graph modification* that realizes both original graph modifications simultaneously. Note that the merge construction in Definition 5 corresponds to the construction in [10].

Definition 5 (Merging Conflict-Free Graph Modifications). *Given parallel independent graph modifications $G \xrightarrow{D_i} H_i$ ($i = 1, 2$). Then, graph modification $G \xrightarrow{D} H$ given by $G \leftarrow D \rightarrow H$ defined by the diagonals of pushouts (1), (4) in Definition 4 is called merged graph modification of $G \xrightarrow{D_i} H_i$ ($i = 1, 2$).*

In [4], we show that in case of parallel independence of $G \xrightarrow{D_i} H_i$ ($i = 1, 2$), the merged graph modification $G \xrightarrow{D} H$ is behaviourally equivalent to the parallel composition $G + G \xrightarrow{D_1+D_2} H_1 + H_2$ of the original graph modifications $G \xrightarrow{D_i} H_i$ ($i = 1, 2$). This result confirms our intuition that the merged graph modification $G \xrightarrow{D} H$ realizes both graph modifications simultaneously. Moreover, it is equal to the sequential compositions of $G \xrightarrow{D_1} H_1 \xrightarrow{D_3} H$ and $G \xrightarrow{D_2} H_2 \xrightarrow{D_4} H$ in Definition 4.

Example 4. The merged graph modification of the two parallel independent graph modifications $m_1 = (G \leftarrow D_1 \rightarrow H_1)$ and $m_2 = (G \leftarrow D_2 \rightarrow H_2)$ in Figure 4 is given by $m = (G \leftarrow D \rightarrow H)$. Obviously, m realizes both graph modifications m_1 and m_2 simultaneously and is shown in [4] to be behaviourally equivalent to their parallel composition $G + G \xrightarrow{D_1+D_2} H_1 + H_2$ and equal to the sequential compositions $G \xrightarrow{D_1} H_1 \xrightarrow{D_3} H$ and $G \xrightarrow{D_2} H_2 \xrightarrow{D_4} H$.

3 Conflict Resolution

If two graph modifications have conflicts, a merge construction according to Definition 5 is not possible any more. In this section, we propose a general merge construction that resolves conflicts by giving *insertion* priority over *deletion* in case of delete-insert conflicts. The result is a merged graph modification where the changes of both original graph modifications are realized as far as possible. We state the properties of the general merge construction and show that the merge construction for the conflict-free case is a special case of the general merge construction.

Definition 6 (Conflicts of Graph Modifications)

1. Two modifications $m_i = G \xrightarrow{D_i} H_i$ ($i = 1, 2$) are conflict-free if they are parallel independent (i.e. we have four pushouts according to Definition 4).
2. They are in conflict if they are not parallel independent.
3. They are in delete-delete conflict if $\exists x \in (G \setminus D_1) \cap (G \setminus D_2)$.
4. (m_1, m_2) are in delete-insert conflict if

$$\begin{aligned} \exists \text{ edge } e \in H_2 \setminus D_2 \text{ with } s(e) \in D_2 \cap (G \setminus D_1) \\ \text{or } t(e) \in D_2 \cap (G \setminus D_1). \end{aligned}$$

Example 5. Consider the graph modifications $m_1 = G \leftarrow D_1 \rightarrow H_1$ and $m_2 = G \leftarrow D_2 \rightarrow H_2$ in Figure 2. (m_2, m_1) are in *delete-insert* conflict because m_2 deletes node S3 which is needed by m_1 for the insertion of an edge. Moreover, m_1 and m_2 are in *delete-delete* conflict because the edge from S1 to S3 is deleted by both m_1 and m_2 . (m_1, m_2) are not in *delete-insert* conflict.

If two modifications m_1 and m_2 are in conflict, then at least one conflict occurs which can be of the following kinds: (1) both modifications delete the same graph element, (2) m_1 deletes a node which shall be source or target of a new edge inserted by m_2 , and (3) m_2 deletes a node which shall be source or target of a new edge inserted by m_1 . Of course, several conflicts may occur simultaneously. In fact, all three conflict situations may occur independently of each other. For example, (m_1, m_2) may be in *delete-delete* conflict, but not in *delete-insert* conflict, or vice versa.⁵

Theorem 1 characterizes the kinds of conflicts that parallel dependent graph modifications may have.

Theorem 1 (Characterization of Conflicts of Graph Modifications).

Given $m_i = (G \xrightarrow{D_i} H_i)$ ($i = 1, 2$), then (m_1, m_2) are in conflict iff

1. (m_1, m_2) are in *delete-delete* conflict, or
2. (m_1, m_2) are in *delete-insert* conflict, or
3. (m_2, m_1) are in *delete-insert* conflict.

⁵ In the worst case, we may have all kinds of conflicts simultaneously.

Proof Idea. Parallel independence of (m_1, m_2) is equivalent to the fact that (PB_1) is also pushout, and the pushout complements (POC_1) and (POC_2) exist, such that pushout (PO_3) can be constructed. By negation, statements 1. - 3. are equivalent to 4. - 6., respectively:

$$\begin{array}{ccccc} G & \xleftarrow{\quad} & D_1 & \xrightarrow{\quad} & H_1 \\ \uparrow (PB_1) & & \uparrow (POC_1) & & \uparrow \\ D_2 & \xleftarrow{\quad} & D & \xrightarrow{\quad} & D_3 \\ \downarrow (POC_2) & & & & \downarrow (PO_3) \\ H_2 & \xleftarrow{\quad} & D_4 & \xrightarrow{\quad} & H \end{array}$$

4. (PB_1) is not a pushout, i.e. $D_1 \rightarrow G \leftarrow D_2$ is not jointly surjective.
5. The dangling condition for $D \rightarrow D_2 \rightarrow H_2$ is not satisfied.
6. The dangling condition for $D \rightarrow D_1 \rightarrow H_1$ is not satisfied.

The *dangling condition* mentioned in statements 5. - 6. is the one known from DPO graph transformation [3]. It is satisfied by inclusions $D \rightarrow D_i \rightarrow H_i$ ($i = 1, 2$), if $\forall e \in H_i \setminus D_i : (s(e) \in D_i \Rightarrow s(e) \in D) \wedge (t(e) \in D_i \Rightarrow t(e) \in D)$. $s(e)$ and $t(e)$ are called *dangling points*. If the dangling condition is satisfied by $D \rightarrow D_i \rightarrow H_i$, the pushout complement (POC_i) can be constructed. \square

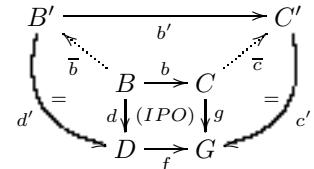
For delete-insert conflicts, our preferred resolution strategy is to preserve the nodes in the merged graph modification that are needed to realize the insertion of edges. If deletion is preferred instead, it has to be done manually after the automatic construction of the merged graph modification, supported by visual conflict indication. Ideally, deletion is done such that predefined meta-model constraints are fulfilled afterwards (see conclusion).

In the following, we define a general merge construction yielding the desired merged graph modification for two given graph modifications with conflicts. In the special case that we have parallel independent graph modifications, it coincides with the conflict-free merge construction in Definition 5.

For the general merge construction, we need so-called *initial pushouts*. In a nutshell, an initial pushout over a graph morphism $f : D \rightarrow G$ extracts a minimal graph morphism $b : B \rightarrow C$ where the context C contains all non-mapped parts of G , and the boundary B consists of those nodes in D that are used for edge insertion (see [3]).

Definition 7 (Initial pushout). Let $f : D \rightarrow G$ be a graph morphism, an initial pushout over f consists of graph morphisms $g : C \rightarrow G$, $b : B \rightarrow C$, and injective $d : B \rightarrow D$ such that f and g are a pushout over b and d .

For every other pushout over f consisting of $c' : C' \rightarrow G$, $b' : B' \rightarrow C'$, and injective $d' : B' \rightarrow D$, there are unique graph morphisms $\bar{b} : B \rightarrow B'$ and $\bar{c} : C \rightarrow C'$ such that $c' \circ \bar{c} = g$ and $d' \circ \bar{b} = d$. Moreover, it is required that (\bar{c}, b') is a pushout over (b, \bar{b}) .



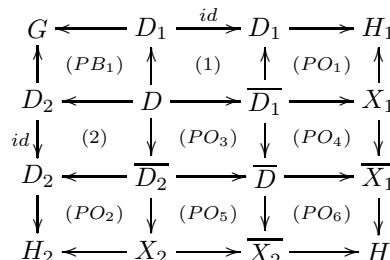
Note that for graph morphisms, there is a canonical construction for initial pushouts [3].

Example 6 (Initial pushout). In the upper right corner of our sample merge construction diagram in Figure 5 below, the initial pushout IPO_1 over the morphism

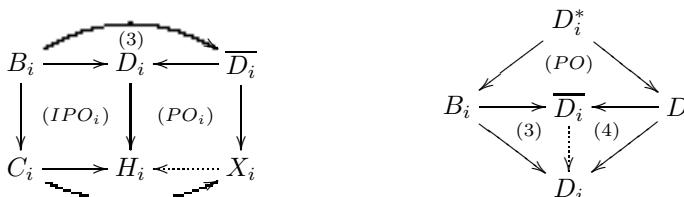
$D_1 \rightarrow H_1$ of graph modification m_1 in Figure 2 is shown. Obviously, the morphism $B_1 \rightarrow C_1$ contains in a minimal context the insertion of the containment edge from S_0 to S_3 .

Now, we are ready to present our general merge construction for graph modifications (see Definition 8). Analogously, to the merging of conflict-free graph modifications we start with constructing the intersection D of the intermediate graphs D_1 and D_2 . In case of delete-insert conflicts where a node is supposed to be deleted by one modification and used as source or target by the other modification, D is too small, i.e. does not contain such nodes. Therefore, we look for a construction which enlarges D to the intermediate graph for the merged modification where insertion is prior to deletion: At first, we identify all these insertions in modifications 1 and 2. This is done by initial pushout construction (as described above) leading to $B_i \rightarrow C_i (i = 1, 2)$. By constructing first the intersection D_i^* of B_i and D in D_i and thereafter the union \overline{D}_i of B_i and D via D_i^* , graph D is extended by exactly those graph elements in B_i needed for insertion later on resulting in \overline{D}_i . After having constructed these extended intermediate graphs \overline{D}_1 and \overline{D}_2 , they have to be glued to result in the intermediate graph \overline{D} of the merged graph modification. Thereafter, the insertions identified by $B_i \rightarrow C_i (i = 1, 2)$ can be transferred to $\overline{D}_i \rightarrow X_i (i = 1, 2)$ first and to $\overline{D} \rightarrow \overline{X}_i (i = 1, 2)$ thereafter. Finally, they are combined by gluing \overline{X}_1 and \overline{X}_2 via \overline{D} yielding result graph H . Since \overline{D} is D extended by graph elements which are not to be deleted, \overline{D} can be embedded into G and thus, can function as intermediate graph for the merged graph modification $G \leftarrow \overline{D} \rightarrow H$.

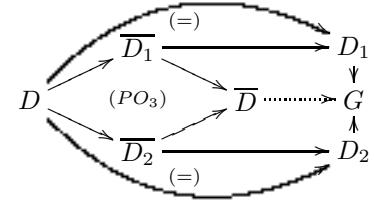
Definition 8 (Merged Graph Modification in General). *Given two graph modifications $G \leftarrow D_1 \rightarrow H_1$ and $G \leftarrow D_2 \rightarrow H_2$. We construct their merged graph modification $G \leftarrow \overline{D} \rightarrow H$ in 6 steps, leading to the following general merge construction diagram:*



1. Construct D by pullback (PB_1) of $D_1 \rightarrow G \leftarrow D_2$.
2. Construct initial pushouts (IPO_i) over $D_i \rightarrow H_i$ for $i = 1, 2$:



3. Construct D_i^* as a pullback of $B_i \rightarrow D_i \leftarrow D$ and \overline{D}_i as pushout of $B_i \leftarrow D_i^* \rightarrow D$ with induced morphism $\overline{D}_i \rightarrow D_i$ with $B_i \rightarrow \overline{D}_i \rightarrow D_i = B_i \rightarrow D_i$ (3) and $D \rightarrow \overline{D}_i \rightarrow D_i = D \rightarrow D_i$ (4) for $(i = 1, 2)$.
4. Construct pushout $\overline{D}_i \rightarrow X_i \leftarrow C_i$ of $\overline{D}_i \leftarrow B_i \rightarrow C_i$ ($i = 1, 2$), leading by (3) to induced morphism $X_i \rightarrow H_i$ and pushout (PO_i) ($i = 1, 2$) by pushout decomposition. Moreover, (4) implies commutativity of (1) and (2) for $(i = 1, 2)$.
5. Now we are able to construct pushouts (PO_3) , (PO_4) , (PO_5) and (PO_6) one after the other.
6. Finally, we obtain the merged graph modification $(G \leftarrow \overline{D} \rightarrow H)$, where $\overline{D} \rightarrow H$ is defined by composition in (PO_6) , and $\overline{D} \rightarrow G$ is uniquely defined as induced morphism using pushout (PO_3) .



Remark 1. If the modifications $m_i = (G \leftarrow D_i \rightarrow H_i)$ ($i = 1, 2$) are parallel independent, then the pullback (PB_1) is a pushout and $\overline{D}_1 = D = \overline{D}_2 = \overline{D}$. In this case, the general merged modification $m = (G \leftarrow \overline{D} \rightarrow H)$ is equal up to isomorphism to the merged graph modification in the conflict-free case in Definition 5. If $m_i = (G \leftarrow D_i \rightarrow H_i)$ ($i = 1, 2$) are in delete-delete conflict, then the merged graph modification deletes the items that are deleted by both m_1 and m_2 since these items are not in D and hence not in \overline{D} .

Example 7. We construct the merged graph modification for graph modifications $m_1 = G \leftarrow D_1 \rightarrow H_1$ and $m_2 = G \leftarrow D_2 \rightarrow H_2$ in Figure 2. The construction diagram is shown in Figure 5.

According to step 3 in Definition 8, \overline{D}_1 has to be constructed as pushout of $B_1 \leftarrow D_1^* \rightarrow D$. D_1^* is the pullback of $B_1 \rightarrow D_1 \leftarrow D$, hence D_1^* consists just of the single node S0. Since B_1 contains two single nodes, S0 and S3, we get as result of step 3 graph \overline{D}_1 which is similar to D but contains additionally node S3. Since (m_1, m_2) are not in delete-insert conflict, $\overline{D}_2 = D$. All remaining squares are constructed as pushouts.

Note that the resulting merged graph modification $G \leftarrow \overline{D} \rightarrow H$ preserves node S3 because this node is deleted in m_2 although it is used for inserting a new edge in m_1 (resolution of the delete-insert conflict). The edge from S1 to S3 is deleted by the merged graph modification as it is deleted by both m_1 and m_2 (resolution of the delete-delete conflict). All graph objects created by either m_1 or m_2 are created also by the merged graph modification. Note that square (2) is a pushout in this example since (m_1, m_2) are not in delete-insert conflict.

The following theorem states that the modification resulting from the general merge construction specifies the intended semantics resolving delete-insert conflicts by preferring insertion over deletion:

Theorem 2 (Behaviour Compatibility of the General Merge Construction). *Given graph modifications $m_i = G \xrightarrow{D_i} H_i$ ($i = 1, 2$) with merged graph*

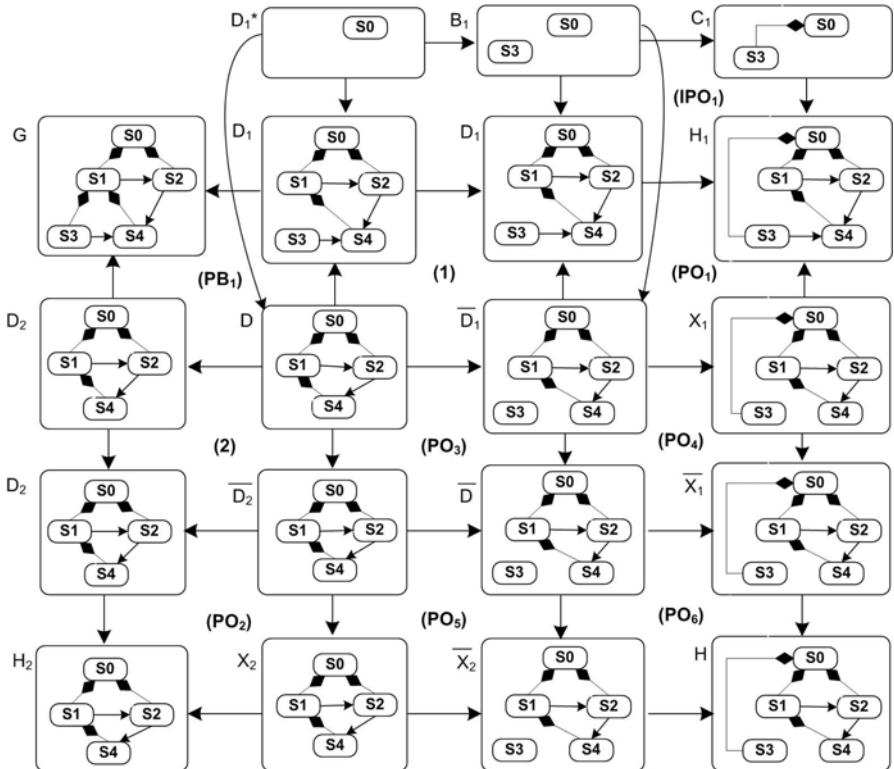


Fig. 5. General merge construction for conflicting graph modifications m_1 and m_2

modification $m = G \xrightarrow{\overline{D}} H = (G \leftarrow \overline{D} \rightarrow H)$ in the sense of Definition 8. We use the following terminology for m (and similarly for m_1, m_2):

$$\begin{aligned} x \in G \text{ preserved by } m &\iff x \in \overline{D}, \\ x \in G \text{ deleted by } m &\iff x \in G \setminus \overline{D}, \\ x \in H \text{ created by } m &\iff x \in H \setminus \overline{D}. \end{aligned}$$

Then, m is behaviour compatible with m_1 and m_2 in the following sense:

1. Preservation: $x \in G$ preserved by m_1 and $m_2 \implies x \in G$ preserved by m
 $\implies x \in G$ preserved by m_1 or m_2
2. Deletion: $x \in G$ deleted by m_1 and $m_2 \implies x \in G$ deleted by m
 $\implies x \in G$ deleted by m_1 or m_2
3. Preservation and Deletion: $x \in G$ preserved by m_1 and $x \in G$ deleted by m_2
 $\implies x \in G$ preserved by m , if $x \in \overline{D}_1$ ⁶
 $x \in G$ deleted by m , if $x \notin \overline{D}_1$ ⁷
 (similar for m_1, m_2, \overline{D}_1 replaced by m_2, m_1, \overline{D}_2)

⁶ In this case, x is a node needed as source or target for an edge inserted by m_1 .

⁷ In this case, x is not needed for edge insertion by m_1 .

4. Creation: $x \in H_1$ created by m_1 or $x \in H_2$ created by m_2
 $\iff x \in H$ created by m

Proof Idea. The preservation, deletion and creation results follow from the pushout properties of \overline{D} , the pushout complement properties of \overline{D}_1 , \overline{D}_2 and the fact that \overline{D}_1 is pullback in the diagrams (PO_1) and (PO_4) (and analogously for \overline{D}_2).

Theorem 3 characterizes the three forms of conflict resolution which may occur.

Theorem 3 (Conflict Resolution by General Merge Construction). *Given graph modifications $m_i = G \xrightarrow{D_i} H_i$ ($i = 1, 2$) that are in conflict. The merge construction $m = (G \leftarrow \overline{D} \rightarrow H)$ resolves the conflicts in the following way:*

1. If (m_1, m_2) are in delete-delete conflict, with both m_1 and m_2 deleting $x \in G$, then x is deleted by m .
2. If (m_1, m_2) are in delete-insert conflict, there is an edge e_2 created by m_2 with $x = s(e_2)$ or $x = t(e_2)$ preserved by m_2 , but deleted by m_1 . Then x is preserved by m .
3. If (m_2, m_1) are in delete-insert conflict, there is an edge e_1 created by m_1 with $x = s(e_1)$ or $x = t(e_1)$ preserved by m_1 , but deleted by m_2 . Then x is preserved by m .

Proof Idea. The resolution of delete-delete conflicts follows from the deletion property, and the resolution of delete-insert conflicts follows from the preservation-deletion property of the general merge construction in Theorem 2.

4 Related Work

First of all, we have to clarify that model merging differs from merging of model modifications. Model merging as presented e.g. in [7,9] is concerned with a set of models and their inter-relations expressed by binary relations. In contrast, merging of model modifications takes change operations into account. Merging of model modifications usually means that non-conflicting parts are merged automatically, while conflicts have to be resolved manually. In the literature, different resolution strategies which allow at least semi-automatic resolution are proposed. A survey on model versioning approaches and especially on conflict resolution strategies is given in [1].

A category-theoretical approach formalizing model versioning is given in [8]. Similar to our approach, modifications are considered as spans of morphisms to describe a partial mapping of models. Merging of model changes is defined by pushout constructions. However, conflict resolution is not yet covered by this approach in a formal way. A category theory-based approach for model versioning in-the-large is given in [2]. However, this approach is not concerned with formalizing conflict resolution strategies. A set-theoretic definition of EMF model merging is presented in [12], but conflicts are solved by the user and not automatically.

In [5] the applied operations are identified first and grouped into parallel independent subsequences then. Conflicts can be resolved by either (1) discarding complete subsequences, (2) combining conflicting operations in an appropriate way, or (3) modifying one or both operations. The choice of conflict resolution is made by the modeler. These conflict resolution strategies have not been formalized. The intended semantics is not directly investigated but the focus is laid on the advantage of identifying compound change operations instead of elementary ones. In contrast, we propose a semi-automatic procedure where at first, an automatic merge construction step gives insertion priority over deletion in case of delete-insert conflicts. If other choices are preferred, the user may perform deletions manually in a succeeding step.

Automatic merge results may not always solve conflicts adequately, especially state-based conflicts or inconsistencies may still exist or arise by the merge construction. Resolution strategies such as resolution rules presented in [6] are intended to solve state-based conflicts or inconsistencies. They can be applied in follow-up graph transformations after the general conflict resolution procedure produced a tentative merge result.

5 Conclusions and Future Work

In this paper, we have formalized a conflict resolution strategy for operation-based conflicts based on graph modifications. Our main result is a general merge construction for conflicting graph modifications. The merge construction realizes a resolution strategy giving insertion priority over deletion in case of delete-insert conflicts to get a merged graph modification result containing as much information as possible. We establish a precise relationship between the behaviour of the given graph modifications and the merged modification concerning deletion, preservation and creation of graph items. In particular, our general merge construction coincides with the conflict-free merge construction if the graph modifications are parallel independent. We show how different kinds of conflicts of given graph modifications are resolved by our automatic resolution strategy. It is up to an additional manual graph modification step to perform deletions that are preferred over insertions.

In [10], we presented two kinds of conflicts which can be detected based on graph modification: *operation-based* and *state-based* conflicts. Hence, in future work, our strategy for solving operation-based conflicts shall be extended by resolving also state-based conflicts. Here, repair actions should be provided to be applied manually by the modeler. Their applications would lead to additional graph modifications optimizing the merged graph modification obtained so far. For the specification of repair actions in this setting, the work by Mens et al. in [6] could be taken into account.

With regard to tool support, our graph transformation environment AGG [11] supports conflict analysis for graph rules and graph modifications. We plan to implement also the check of behavioural equivalence and the general merge

construction for graph modifications in near future. This proof-of-concept implementation could function as blueprint for implementing our new resolution strategy in emerging model versioning tools.

References

1. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. *IJWIS* 5(3), 271–304 (2009)
2. Diskin, Z., Czarnecki, K., Antkiewicz, M.: Model-versioning-in-the-large: Algebraic foundations and the tile notation. In: Proc. of Workshop on Comparison and Versioning of Software Models (CVSM 2009), pp. 7–12. IEEE Computer Society, Los Alamitos (2009)
3. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. In: EATCS Monographs in Theor. Comp. Science. Springer, Heidelberg (2006)
4. Ehrig, H., Ermel, C., Taentzer, G.: A formal resolution strategy for operation-based conflicts in model versioning using graph modifications: Extended version. Tech. rep., TU Berlin (to appear, 2011)
5. Küster, J.M., Gerth, C., Engels, G.: Dependent and conflicting change operations of process models. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 158–173. Springer, Heidelberg (2009)
6. Mens, T., van der Straeten, R., D'Hondt, M.: Detecting and resolving model inconsistencies using transformation dependency analysis. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 200–214. Springer, Heidelberg (2006)
7. Pottinger, R., Bernstein, P.A.: Merging models based on given correspondences. In: Aberer, K., Koubarakis, M., Kalogeraki, V. (eds.) VLDB 2003. LNCS, vol. 2944, pp. 826–873. Springer, Heidelberg (2004)
8. Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A category-theoretical approach to the formalisation of version control in MDE. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 64–78. Springer, Heidelberg (2009)
9. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S.M., Chechik, M.: Consistency checking of conceptual models via model merging. In: Proc. IEEE Int. Conf. on Requirements Engineering, pp. 221–230. IEEE, Los Alamitos (2007)
10. Taentzer, G., Ermel, C., Langer, P., Wimmer, M.: Conflict detection for model versioning based on graph modifications. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 171–186. Springer, Heidelberg (2010)
11. TFS-Group, TU Berlin: AGG (2009), <http://tfs.cs.tu-berlin.de/agg>
12. Westfechtel, B.: A formal approach to three-way merging of EMF models. In: Proc. Workshop on Model Comparison in Practice (IWMCP), pp. 31–41. ACM, New York (2010)

A Step-Wise Approach for Integrating QoS throughout Software Development

Stéphanie Gatti, Emilie Balland, and Charles Consel

Thales Airborne Systems / University of Bordeaux / INRIA, France
`first-name.last-name@inria.fr`

Abstract. When developing real-time systems such as avionics software, it is critical to ensure the performance of these systems. In general, deterministic Quality of Service (QoS) is guaranteed by the execution platform, independently of a particular application. For example, in the avionics domain, the ARINC 664 standard defines a data network that provides deterministic QoS guarantees. However, this strategy falls short of addressing how the QoS requirements of an application get transformed through all development phases and artifacts. Existing approaches provide support for QoS concerns that only cover part of the development process, preventing traceability.

In this paper, we propose a declarative approach for specifying QoS requirements that covers the complete software development process, from the requirements analysis to the deployment. This step-wise approach is dedicated to control-loop systems such as avionics software. The domain-specific trait of this approach enables the stakeholders to be guided and ensures QoS requirements traceability via a tool-based methodology.

Keywords: Quality of Service, Domain-Specific Design Language, Tool-Based Development Methodology, Generative Programming.

1 Introduction

Non-functional requirements are used to express the *quality* to be expected from a system. For real-time systems such as avionics, it is critical to guarantee this quality, in particular time-related performance properties. For example, the avionics standard ARINC 653 defines a Real-Time Operating System (RTOS) providing deterministic scheduling [3] and thus ensuring execution fairness between applications. Another example is the ARINC 664 that defines Avionics Full DupleX switched Ethernet (AFDX), a network providing deterministic Quality of Service (QoS) for data communication [4]. In this domain, deterministic QoS is generally ensured at the execution platform level (*e.g.*, operating systems, distributed systems technologies, hardware specificities), independently of a particular application. When addressing the QoS requirements of a given application, these platform-specific guarantees are not sufficient.

There exist numerous specification languages to declare QoS requirements at the architectural level [1]. Initially, these languages were mostly contemplative. Several recent approaches also provide support to manage specific aspects

(*e.g.*, coherence checking [10], prediction [20], monitoring [17]). These approaches are generally dedicated to a particular development stage, leading to a loss of traceability (*i.e.*, the ability to trace all the requirements throughout the development process). In the avionics certification processes [11,12,5], traceability is mandatory for both functional and non-functional requirements. The functional traceability is usually ensured by systematic development methodologies such as the V-model that guides stakeholders from the requirements analysis to the system deployment. Similarly, QoS should be fully integrated into the development process as it is a crosscutting concern [19].

In this paper, we propose a step-wise QoS approach integrated through all development phases and development artifacts. This approach is dedicated to control-loop systems. Control-loop systems are systems that sense the external environment, compute data, and eventually control the environment accordingly. This kind of systems can be found in a range of domains, including avionics, robotics, and pervasive computing. For example, in the avionics domain, a flight management application is a control-loop system that (1) senses the environment for location and other navigation information, (2) computes the trajectory and (3) modifies the wings configuration accordingly. The contributions of this paper can be summarized as follows.

A step-wise QoS approach dedicated to control-loop systems. We propose a step-wise approach that systematically processes QoS requirements throughout software development. This integrated approach is dedicated to control-loop systems, allowing to rely on a particular architectural pattern and thus enhancing the design and programming support level for non-functional aspects. In this paper, we focus on time-related performance but the approach could be generalized to other non-functional properties (*e.g.*, CPU or memory consumption).

Requirements Traceability. In the avionics domain, the traceability of both functional and non-functional requirements is critical [11]. In our approach, the traceability is ensured by the systematic propagation of constraints derived from the QoS declarations and applied to each development step.

A tool-based methodology. Our approach has been integrated into DiaSuite, a tool-based development methodology dedicated to control-loop systems [8]. DiaSuite is based on a dedicated design language that we have enriched with time-related performance properties. This non-functional extension has been used to offer verification and programming support at each development stage.

Experiments in the avionics domain. Our approach has been applied to the development of various avionics applications, including a flight management system and a collision avoidance system. These experiments have demonstrated that our step-wise approach can effectively guide the avionics certification process.

2 Background and Working Example

This section presents a working example used throughout this paper. This presentation is done in the context of the DiaSuite development methodology [8].

We choose a control-loop system from the avionics domain: a simplified version of an aircraft guidance application, controlling the trajectory of an aircraft by correcting the configurations of ailerons.

2.1 Overview of the DiaSuite Approach

The DiaSuite approach is a tool-based methodology dedicated to control-loop systems. DiaSuite provides support for each development stage (from design to deployment) as depicted in Figure 1.

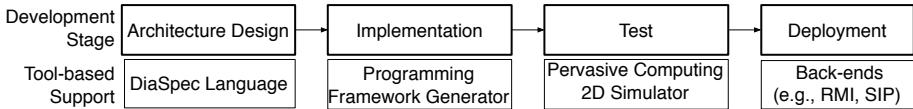


Fig. 1. The DiaSuite tool-based development process

During the design stage, the DiaSpec language allows to design an application using an architectural pattern dedicated to control-loop systems. This specific architectural pattern comprises four layers of components: (1) *sensors* obtain raw data from the environment; (2) *contexts* process data and provide high-level information; (3) *controllers* use this information to control actuators; (4) *actuators* impact the environment. The sensors and actuators are the two facets of *entities* corresponding to devices, whether hardware or software, deployed in an environment.

This specification guides the developer throughout the development process. The DiaSpec compiler generates a Java programming framework dedicated to the application. This framework precisely guides the programmer during the implementation stage by providing high-level operations for entity discovery and component interactions. Based on these declarations, a simulator dedicated to pervasive computing environments is used to test and simulate the system. Then, the DiaSuite back-ends enable the deployment of an application by targeting a specific distributed systems technology such as RMI, SIP or Web Services.

2.2 Aircraft Guidance Application

The aircraft guidance application uses two sensors for computing the actual aircraft trajectory: the inertial reference unit, providing the localization, and the air data unit, supplying such measurements as the airspeed and the angle of attack. The synchronization of both information sources allows to compute the actual aircraft trajectory. This trajectory is then compared to the flight plan entered by the pilot and used for controlling and correcting ailerons by the automatic pilot, if necessary.

Following the DiaSuite development methodology, the first step identifies the devices involved in the aircraft guidance application using a domain-specific taxonomy of entities, as can be found in the aeronautics literature. In this example,

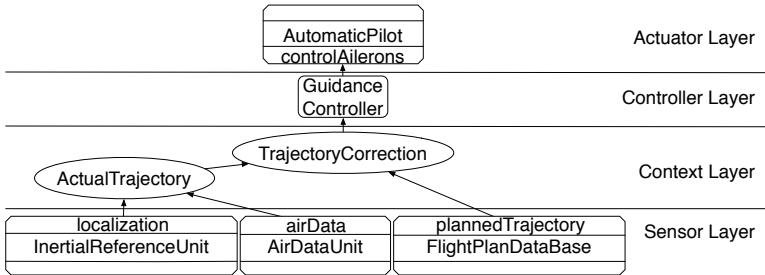


Fig. 2. A data-flow view of the aircraft guidance application

we have identified four entities: the inertial reference unit, the air data unit, the flight plan database and the automatic pilot. The second step of the methodology consists of designing the application using DiaSpec. The system description is illustrated in Figure 2, making explicit the four component layers of DiaSpec.

In the example, the `InertialReferenceUnit` sensor provides the current localization of the aircraft. The `AirDataUnit` sensor supplies several air data such as the airspeed and the angle of attack. All these data are sent to the `ActualTrajectory` context that is responsible for computing the current trajectory of the aircraft. This information is then sent to the `TrajectoryCorrection` context component. When receiving a new trajectory, the `TrajectoryCorrection` component gets the planned trajectory (from the flight plan initially entered by the pilot) from the `FlightPlanDatabase` component. By comparing these information sources, it computes trajectory corrections that are sent to the `GuidanceController` component, responsible for controlling ailerons through the `AutomaticPilot` actuator.

The avionics certification process requires this trajectory readjustment to be time-bounded. In the next section, we show how the DiaSuite approach, enriched with time-related properties, can guide the development of such critical applications.

3 QoS throughout Software Development

This section presents how QoS requirements can be systematically processed throughout software development.

3.1 Requirements Analysis and Functional Specification

In software development methodologies, the requirements analysis stage identifies the users' needs. Then, the functional specification stage identifies the main functionalities to be fulfilled by the application to satisfy the users' requirements. In the avionics domain, each of these functionalities is generally associated to a *functional chain* [26], representing a chain of computations, from sensors to actuators.

The aircraft guidance system has a unique functional chain, whose execution should take less than 3 seconds, according to our expert at Thales Airborne Systems. In avionics, such time constraints, directly associated to a specific functional chain, is referred to as Worst Case Execution Time (WCET)¹. If the design process involves refinement steps such as the identification of functional chain segments, the WCETs can be further refined. For example, we can identify a functional chain segment corresponding to the computation of the actual trajectory (from the sensors feeding the *ActualTrajectory* context). This chain segment can be reused in other applications, *e.g.*, displaying the actual trajectory on the navigation display unit. According to our expert, this chain segment must not take more than 2 seconds to execute.

3.2 Architecture Design

During the architecture design stage, the functional chains are decomposed into connected components. The architect can then refine the WCET of the functional chain on time-related constraints at the component level.

In the DiaSuite architectural pattern, the data flow between two components can be realized using two interaction modes: by pulling data (one-to-one synchronous interaction mode with a return value) or by pushing data to event subscribers (asynchronous publish/subscribe interaction mode). Pull interactions are typically addressed by a *response time* requirement as is done in Web Services [17]. Push interactions raise a need to synchronize two or more input events of a component. This need is addressed by introducing a *freshness* requirement on the input event values. This requirement is in the spirit of synchronization skew in the multimedia domain [18]. In addition to the freshness constraint, we define the *bounded synchronization-time* constraint that authorizes desynchronization during a bounded time, avoiding diverging synchronization strategies.

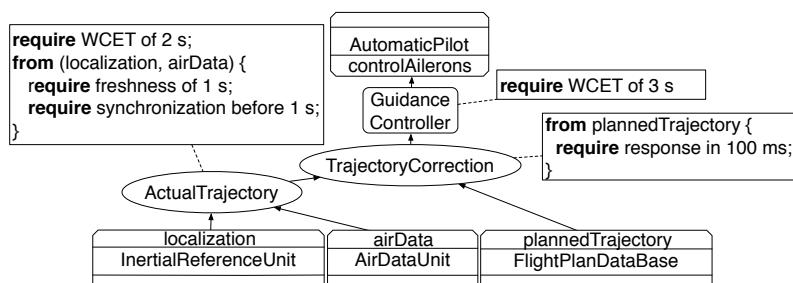


Fig. 3. Architecture of the working example, enriched with QoS contracts

Figure 3 shows the QoS contracts associated to each component in the flight guidance application. The WCETs associated to the functional chain of the

¹ This usage of WCET is only loosely related to the notion of WCET as documented in the literature for hard real-time systems.

aircraft guidance and to the trajectory computing chain segment are mapped to the `GuidanceController` and `ActualTrajectory` components, respectively. The WCET of the functional chain of the aircraft guidance is refined into (1) a freshness constraint of 1 second between `localization` and `airData` with an equal bounded synchronization-time constraint (since the WCET is not compatible with a longer desynchronization time); and (2) a response time of 100 milliseconds of `FlightPlanDataBase`. The WCET associated to the functional chain of the aircraft guidance is translated into a QoS contract, attached to `GuidanceController` as controllers are generally dedicated to a given functional chain.

The QoS contracts are introduced as an extension of DiaSpec. Figure 4 shows an extract from the DiaSpec specification.

```
context ActualTrajectory as Trajectory {
    source localization from InertialReferenceUnit;
    source airData from AirDataUnit;
    qos {
        from (localization, airData) {
            require freshness of 1 s;
            require synchronization before 1 s;
        }
    }
}
```

Fig. 4. DiaSpec declaration of the `ActualTrajectory` component

The `ActualTrajectory` component is declared with the `context` keyword, and returns values of type `Trajectory`. This component processes two sources of information: `localization` and `airData`. These sources are declared using the `source` keyword that takes a source name and a class of entities. Then, the QoS contract declared using the `qos` keyword defines freshness and synchronization constraints between `localization` and `airData`. This domain-specific approach guides the stakeholders when adding QoS requirements by automatically enforcing the conformance between the QoS contracts and the functional constraints.

3.3 Implementation

The DiaSuite approach includes a compiler that generates a dedicated programming framework from a DiaSpec description. Our approach enriches this process by generating runtime-monitoring support from QoS declarations. At the implementation level, QoS requirements on components become runtime verifications that rely on the communication methods of the generated programming framework. Monitoring mechanisms are encapsulated into component containers that ensure that the response time and the freshness requirements are respected. The approach based on containers allows a separation of concerns between functional and non-functional requirements because a container is only in charge of

intercepting calls for monitoring requirements, and forwarding the calls to the functional component. If a QoS contract is violated, the container throws specific exceptions `ResponseTimeException` or `SynchronizationException`. The treatment of such exceptions is left to the developer. It may involve any number of actions, including logging or reconfiguration [17]. DiaSuite provides declarative support at the architectural level to design exceptional treatments [21], preventing the application to be bloated and entangled with error-handling code.

The code corresponding to the response time requirement is straightforward. It is based on a timer that calculates the elapsed time between the request and the response. The more elaborate part concerns the synchronization defined by the automaton depicted in Figure 5. Suppose we want to synchronize `data1` and `data2` values. When receiving the first data (`S2` and `S4` states), the container activates the t and t' timers for measuring respectively synchronization time and freshness ($t, t' := 0$). While synchronization time is not reached ($t \leq ts$), the container waits for fresh data. If the other data is received before the freshness time has elapsed, the container pushes both data to the functional component (`S5` final state). Otherwise, if the freshness is not respected ($t' > tf$), the data is rejected. This is not considered as an error state since we authorize desynchronization for a finite period. Thereby the container waits for new values (`S3` state). If the synchronization time has elapsed ($t > ts$), the synchronization is aborted and a `SynchronizationException` is thrown (error state).

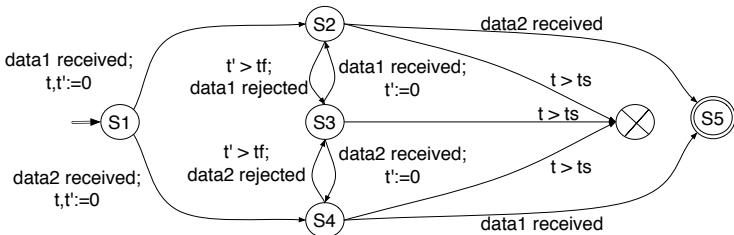


Fig. 5. Synchronization automaton

3.4 Deployment

Our approach offers support for predicting the performance of an application by injecting deployment parameters, such as distributed systems technologies, platform and hardware characteristics. By taking advantage of their QoS characteristics (*e.g.*, the guaranteed deterministic timing of the AFDX network [4]), it is possible to refine the time-related requirements generated from the QoS declarations, and thus to compare several deployment configurations. In particular, it allows technologies to be selected according to their time-related properties.

This prediction tool takes numerical constraints generated from the QoS declarations as input. Then, an external constraint solver [9] checks whether a configuration respects the WCET of the functional chain and predicts constraints

to the other architectural elements. In the next section, we detail how these numerical constraints are generated and propagated throughout the software development process.

4 QoS Requirements Traceability

The requirements traceability is the guarantee for each requirement to be traced back to its origin (*i.e.*, a QoS declaration), by following its propagation in the software development process. By generating numerical constraints from the QoS declarations, our step-wise approach allows the traceability of QoS requirements during software development. This approach is summarized in Figure 6.

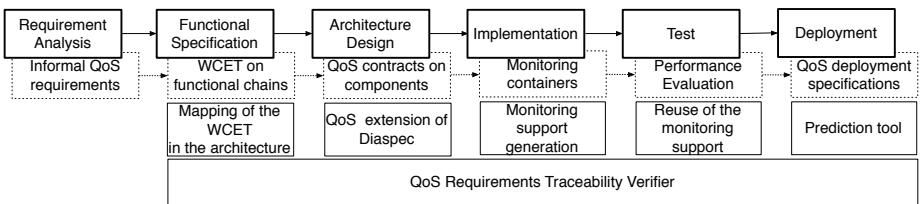


Fig. 6. Development process of Figure 1 extended with QoS concerns

At each development step, QoS declarations are translated into numerical constraints that are fed to the verifier of requirements traceability. This verifier propagates these numerical constraints between the development stages, and checks whether no new constraint invalidates constraints from preceding stages. In this section, we detail how these numerical constraints are generated and propagated.

4.1 From Functional Specification to Architecture Design

From functional specification to architecture design, requirements on functional chains (or chain segments) are refined into requirements on components. By generating numerical constraints, it is possible to ensure that the refinement does not invalidate requirements from the previous stage. Doing so amounts to checking whether the constraint system is still satisfied. The generation of these numerical constraints is inspired by the Defour *et al.*'s work [10] and relies on the DiaSuite architectural pattern. It consists of automatically translating the QoS contracts presented in Section 3 into numerical equations specifying time relationships between the components. For example, let us detail how these equations are generated for the aircraft guidance application.

WCET on functional chains and chain segments. The functional chain that controls the trajectory has a WCET of 3 seconds, represented by the contract attached to `GuidanceController`. This leads to the following numerical constraints:

```

T_wcet_GuidanceController <= 3;
T_wcet_GuidanceController =
    T_provide_GuidanceController +
    T_com(GuidanceController, AutomaticPilot);

```

The first equation represents the WCET associated to `GuidanceController`. The second equation refines this functional chain into a sequence of two functions: one for computing orders (the `GuidanceController` chain segment), and one for communicating these orders to the automatic pilot. Thus, the global time is the sum of the `T_provide_GuidanceController` time and the communication time between `GuidanceController` and `AutomaticPilot` (denoted by the `T_com` function). The `T_provide_GuidanceController` time corresponds to the chain segment between the moment when the `InertialReferenceUnit` or `AirDataUnit` sensor sends a value and the moment when `GuidanceController` issues orders to `AutomaticPilot`.

Similarly, the `T_provide_GuidanceController` time can also be refined into the time of `GuidanceController` to compute orders, the communication time between `TrajectoryCorrection` and `GuidanceController`, and the global time associated to the chain segment of `TrajectoryCorrection`:

```

T_provide_GuidanceController =
    T_provide_TrajectoryCorrection +
    T_com(TrajectoryCorrection, GuidanceController) +
    T_compute_GuidanceController;

```

Response Time. The time associated to `TrajectoryCorrection` can be refined with respect to its relation with `ActualTrajectory` and `FlightPlanDataBase`:

```

T_provide_TrajectoryCorrection =
    T_provide_ActualTrajectory +
    T_com(ActualTrajectory, TrajectoryCorrection) +
    2 * T_com(FlightPlanDataBase, TrajectoryCorrection) +
    T_provide_FlightPlanDataBase +
    T_compute_TrajectoryCorrection;

```

Between the `ActualTrajectory` and `TrajectoryCorrection` contexts, the communication mode is of type publish/subscribe and thus can be decomposed into `T_provide_ActualTrajectory` corresponding to the chain segment for computing the trajectory and `T_com(ActualTrajectory, TrajectoryCorrection)` corresponding to the communication time between these two contexts. Because the `plannedTrajectory` source of the `FlightPlanDataBase` is accessed by pulling the value in a synchronous manner, it is decomposed into the time to compute the data (`T_provide_FlightPlanDataBase`) and the communication round-trip (`2 * T_com(FlightPlanDataBase, TrajectoryCorrection)`) between the two components, assuming the size of the request and the response fit within a MTU (Maximum Transmission Unit).

Freshness and Bounded Synchronization Time. The QoS contract associated to `ActualTrajectory` specifies freshness and bounded synchronization time

between `InertialReferenceUnit` and `AirDataUnit`. In the worst case, the synchronization takes the sum of the bounded synchronization time and the maximum time for receiving an event from `InertialReferenceUnit` or `AirDataUnit`. This leads to the generation of the following constraints:

```
T_provide_ActualTrajectory <= 2;
T_synchronization <= 1;
T_provide_ActualTrajectory =
max(
  T_provide_airData + T_com(AirDataUnit,ActualTrajectory),
  T_provide_localization + T_com(InertialReferenceUnit,ActualTrajectory)) +
T_synchronization +
T_compute_ActualTrajectory;
```

The refinement of the numerical constraints and the checking of coherence at each step ensures the coherence between all non-functional requirements. Each numerical constraint is defined using Prolog IV [9], a constraint logic programming language over real numbers, coupled with a real interval arithmetic solver for checking the coherence of each refinement step.

4.2 From Architecture Design to Implementation

Monitoring support is generated from the non-functional specifications. Each non-functional container is in charge of monitoring the time-related constraints associated to a given functional component. Since QoS declarations at the design level are used to generate the monitoring support, the traceability is automatically ensured between the design and implementation stages. Moreover, as this support is embedded into the programming framework, it is completely transparent to the developer. Doing so prevents the developer from introducing errors in the monitoring code. Specifically the DiaSuite exception mechanism allows to separate the detection mechanism that is generated from the treatment code that is implemented by the developer.

4.3 From Implementation to Deployment

During the deployment stage, the prediction tool is based on the numerical constraints generated from the QoS-extended DiaSpec specification, ensuring the traceability of the requirements. In the avionics domain, the execution platforms offer deterministic QoS characteristics (*e.g.*, the deterministic timing of the AFDX network). Such information allows to refine the time-related constraints generated from a QoS declaration and to compare the performance of several deployment configurations.

In the generated constraints, there are several numerical variables that depend on the deployment. The `T_com_<component_name>` variables depend on the communication mode. If the application is deployed on a non-distributed platform, the communication time can be considered as null between applicative components, simplifying the numerical constraints. If the application executes

on a distributed platform, the communication time between the applicative components depends on the distributed systems technologies. In avionics, the most commonly used network is the AFDX. The communication constraints can be refined according to the AFDX bandwidth and the associated Bandwidth Allocation Gap (BAG). Concerning the communication between applicative components and devices, different sort of communication technologies can be used, such as a serial link (*e.g.*, ARINC429, RS422) that leads to different communication times. The `T_compute_<component_name>` variables depend on the complexity of the algorithm and the execution platform (*e.g.*, CPU frequency and memory access time). Finally, the `T_provide_<data_sensed>` variables depend on sensor technologies (*e.g.*, mechanical or LASER probes for air data).

For example, assume we enrich the system with a new functionality for displaying the trajectory on the navigation display. We want to reuse the chain segment computing the actual trajectory but with stronger QoS requirements, leading to the constraint `T_provide_ActualTrajectory <= 0.8`. From all these constraints, the prediction tool infers the following value range for the Air Data Unit: `0 <= T_provide_airData <= 0.8`. This constraint is propagated to the stakeholders in charge of selecting the technologies for the execution platform. In this situation, they will choose LASER probes whose performance is conform with this constraint.

5 Towards Certification of Avionics Systems

Our approach has been applied to the design of several avionics applications, including the flight management system, the aircraft guidance system, and the traffic collision avoidance system. These experiments have shown that the Dia-Suite methodology is well-suited for the development of avionics control-loop systems. In this section, we discuss how our integrated QoS approach guides the avionics certification process.

In avionics, aircrafts have to respect the Certification Specification (CS) standard to obtain the airworthiness certificate. CS proposes the classification of the failure conditions: conditions impacting the aircraft and/or its occupants. They depend on the flight phases (*e.g.*, landing) and the environmental conditions. Failure conditions are associated with a Design Assurance Level, indicating their degree of criticality and safety objectives. To reach safety objectives, aeronautical standards specify constraints on the development process (*e.g.*, distribute the development stages across several teams) and the testing process (*e.g.*, structural and black-boxes tests). These constraints cover all the levels of a system, including the software and hardware layers. Both functional and non-functional requirements [22] have to be guaranteed, including performance concerns.

Modern avionics platforms such as the Integrated Modular Avionics (IMA) allow to host several functions on the same platform [25]. The IMA approach introduces different stakeholders in the development process. The *system integrator* is the leading authority (generally, the airframer). The role of this stakeholder is to integrate all IMA systems: the software applications, provided by the

function suppliers, and the hardware systems, provided by the *platform suppliers*. The integrator and suppliers play different roles in the certification process [12]. The system integrator specifies general constraints about all the systems at the aircraft level. For example, this stakeholder defines the Worst Case Communication Time (WCCT) of the network, ensured by the AFDX technology [4], and the WCET of the applications. These WCET requirements are passed on to the function suppliers. To fulfill them, in turn, function suppliers produce specific requirements for platform suppliers regarding issues such as time slots, CPU power, memory capacity and throughput. To do so, the platform suppliers have to provide the function suppliers with all the characteristics of this platform, including WCET for core software services (*e.g.*, drivers and health monitoring).

From the high-level WCET constraints delivered by the system integrator, our approach guides the function suppliers in systematically specifying and refining all the non-functional requirements during the development of the application. Function suppliers can also use the generated monitoring support for validating the performance of their application. Furthermore, the prediction tool can be helpful in determining an interval of timing on each deployment technologies and passing these constraints on to the platform suppliers. The platform suppliers are now responsible for giving applications the means to access input data from the sources, and send output data to the actuators. Regarding constraints issued by all the function suppliers, the platform supplier can use the prediction tool for proposing a platform configuration that matches all their needs. Finally, the system integrator can use the generated monitoring support for both controlling the applications in flight, and performing aircraft maintenance. In doing so, information about the non-functional behavior of the application can be logged in flight and the pilot can be alerted in case of unexpected behaviors. The monitoring support can also be used on-ground for correcting errors. As this support is entirely generated, it is sufficient to test and certify the generator for guaranteeing the correctness of all future generated monitoring containers.

Requirements traceability is key to obtain avionics certification. Because the role separation proposed by the IMA platforms leads to the collaboration of several companies, requirements traceability has become significantly more challenging. The tool-based approach proposed in this paper can facilitate this certification process by offering support for propagating automatically QoS requirements between the stakeholders.

6 Related Work

Among existing QoS specifications languages, some of them focus on performance properties and already offer design and programming support. For example, in the spirit of our approach, Defour *et al.* generate numerical constraints from time-based requirements specified with the QoSCL language [10]. These constraints are not only used to check the requirements compatibility according to the architecture, but also to predict the QoS from the number of instances of each component. AlTurki *et al.* propose a real-time rewriting model backing a timing

specification language [2]. It allows them to verify various real-time properties using the Maude rewriting engine. Krogmann *et al.* have set up a quantitative performance prediction tool into the Palladio Component Model [20], allowing architects to choose between different architectural designs. They define many types of performance requirements that would be interesting for us to take into account for critical systems (*e.g.*, CPU). Bertolino *et al.* [6] also propose another performance-based prediction approach that focuses on the assembling of existing components. In contrast to our work, the above approaches are general-purpose and are limited to the design phases of the development process.

Other approaches are domain-specific, for example dedicated to the specification of real-time systems. As a result, they can offer better design and programming support. For example, Fredriksson *et al.* propose a framework for leveraging non-functional requirements (*e.g.*, time and memory consumption) to build control systems components [15] and thus to predict the functional/non-functional behavior of the composed system. Doose *et al.* formalize real-time systems as a set of functionalities linked within timed communications and then verify the time-coherency of the whole system using model-checking techniques [13]. Carcenac *et al.* also validate real-time systems according to the incremental specification of non-functional requirements [7]. Yet, these approaches only focus on the validation of the systems.

In contrast, the approach of Robert *et al.* enables generating monitoring support from non-functional requirements, represented as exceptional transitions in timed-automata [24]. In the same spirit, Duclos *et al.* [14] have proposed to specify QoS requirements as aspects in the architectural models for providing separation of concerns, monitoring these requirements at runtime. The approach of Gessler *et al.* enables generating scheduling support based on QoS declarations [16].

The above QoS approaches are dedicated to real-time systems and offer support at design time (*e.g.*, prediction) and/or at runtime (*e.g.*, monitoring). However, they are mostly dedicated to specific development stages and do not consider the traceability of non-functional concerns through the software development process.

To conclude, specifying non-functional requirements only at the architecture level is not sufficient. As observed by Koziolek *et al.* [19], it is crucial to clearly identify the stakeholders and the workflow between the functional development and the non-functional layer. Towards this end, we propose a unified approach that integrates QoS into the complete development process.

7 Conclusion

In this paper we have presented a step-wise approach integrating QoS concerns through all phases of software development. This approach dedicated to control-loop systems extends the DiaSuite tool-based methodology by offering support for specifying, validating and monitoring time-related requirements. We have shown that this domain-specific approach allows to guide the stakeholders in

systematically refining non-functional requirements and ensures requirements traceability by generating numerical constraints. We have illustrated our approach in the avionics domain where such QoS requirements are critical.

We are currently working on a deeper evaluation of this approach with the development of an autopilot application coupled to the FlightGear simulator [23]. This work will allow to leverage DiaSpec's architectural support of error handling [21] for treating the violation of a QoS contract at runtime. In particular, we plan to show how the error handling support provided by DiaSpec can be used to implement logging and reconfiguration treatments. This evaluation would also help in refining the specification language (*e.g.*, time constraints depending on input parameters). Future work concerns the integration of this methodology into the avionics certification process. In particular, we will need to certify our tools and their associated development approach.

References

1. Aagedal, J.Ø.: Quality of service support in development of distributed systems. PhD thesis, University of Oslo (2001)
2. AlTurki, M., Dhurjati, D., Yu, D., Chander, A., Inamura, H.: Formal specification and analysis of timing properties in software systems. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 262–277. Springer, Heidelberg (2009)
3. ARINC 653, system partitioning and scheduling. Aeronautical Radio, Inc. (2003)
4. ARINC 664, AFDX: Avionics Full Duplex switched ethernet. Aeronautical Radio, Inc. (2005)
5. ARP4754, certification considerations for highly-integrated or complex aircraft systems, SAE (1996)
6. Bertolini, A., Mirandola, R.: CB-SPE tool: putting component-based performance engineering into practice. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 233–248. Springer, Heidelberg (2004)
7. Carcenac, F., Boniol, F.: A formal framework for verifying distributed embedded systems based on abstraction methods. International Journal on Software Tools for Technology Transfer 8(6), 471–484 (2006)
8. Cassou, D., Bertran, B., Lorient, N., Consel, C.: A generative programming approach to developing pervasive computing systems. In: Proceedings of the 8th International Conference on Generative Programming and Component Engineering, pp. 137–146. ACM, New York (2009)
9. Colmerauer, A.: Specifications of Prolog IV (1996)
10. Defour, O., Jézéquel, J.-M., Plouzeau, N.: Extra-functional contract support in components. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 217–232. Springer, Heidelberg (2004)
11. DO-178B, software considerations in airborne systems and equipment certification, RTCA, Inc. (1992)
12. DO-297, Integrated Modular Avionics (IMA) development guidance and certification considerations, RTCA, Inc. (2005)
13. Doose, D., Mammeri, Z.: Polyhedra-based approach for incremental validation of real-time systems. In: Yang, L.T., Amamiya, M., Liu, Z., Guo, M., Rammig, F.J. (eds.) EUC 2005. LNCS, vol. 3824, pp. 184–193. Springer, Heidelberg (2005)

14. Duclos, F., Estublier, J., Morat, P.: Describing and using non-functional aspects in component-based applications. In: Proceedings of the 1st International Conference on Aspect-Oriented Software Development, pp. 65–75. ACM, New York (2002)
15. Fredriksson, J., Tivoli, M., Crnkovic, I.: A component-based development framework for supporting functional and non-functional analysis in control system design. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 368–371. ACM, New York (2005)
16. Genssler, T., Christoph, A., Winter, M., Nierstrasz, O., Ducasse, S., Wuyts, R., Arévalo, G., Schönhage, B., Müller, P.O., Stich, C.: Components for embedded software: the PECOS approach. In: Proceedings of the Conference on Compilers, Architectures and Synthesis for Embedded Systems, pp. 19–26. ACM, New York (2002)
17. Halima, R.B., Drira, K., Jmaiel, M.: A QoS-oriented reconfigurable middleware for self-healing web services. In: Proceedings of the 6th IEEE International Conference on Web Services, pp. 104–111. IEEE, Los Alamitos (2008)
18. Jha, S., Seneviratne, A.: Synchronization skew: a QoS measurement study. In: Proceedings of the Conference on Local Computer Networks, pp. 77–78 (1999)
19. Koziolek, H., Happe, J.: A QoS driven development process model for component-based software systems. In: Gorton, I., Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Ren, X.-M., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, pp. 336–343. Springer, Heidelberg (2006)
20. Krogmann, K., Schweda, C.M., Buckl, S., Kuperberg, M., Martens, A., Matthes, F.: Improved feedback for architectural performance prediction using software cartography visualizations. In: Mirandola, R., Gorton, I., Hofmeister, C. (eds.) QoSA 2009. LNCS, vol. 5581, pp. 52–69. Springer, Heidelberg (2009)
21. Mercadal, J., Enard, Q., Consel, C., Lorient, N.: A domain-specific approach to architecturing error handling in pervasive computing. In: Proceedings of the 25th International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. ACM, New York (2010)
22. Paulitsch, M., Rueess, H., Sorea, M.: Non-functional avionics requirements. In: Proceedings of the 3rd International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, pp. 369–384. Springer, Heidelberg (2009)
23. Perry, A.R.: The FlightGear flight simulator. In: Proceedings of the USENIX Annual Technical Conference (2004)
24. Robert, T., Fabre, J.-C., Roy, M.: On-line monitoring of real time applications for early error detection. In: Proceedings of the 14th IEEE Pacific Rim International Symposium on Dependable Computing, pp. 24–31. IEEE, Los Alamitos (2008)
25. Watkins, C.B., Walter, R.: Transitioning from federated avionics architectures to Integrated Modular Avionics. In: Proceedings of the 26th IEEE/AIAA Digital Avionics Systems Conference, p. 2. IEEE, Los Alamitos (2007)
26. Windsor, J., Hjortnaes, K.: Time and space partitioning in spacecraft avionics. In: Proceedings of the 3rd IEEE International Conference on Space Mission Challenges for Information Technology, pp. 13–20. IEEE, Los Alamitos (2009)

Systematic Development of UMLsec Design Models Based on Security Requirements

Denis Hatebur^{1,4}, Maritta Heisel¹, Jan Jürjens^{2,3}, and Holger Schmidt²

¹ Software Engineering, Department of Computer Science and Applied Cognitive Science, Faculty of Engineering, University Duisburg-Essen, Germany
`{denis.hatebur,maritta.heisel}@uni-due.de`

² Software Engineering, Department of Computer Science, TU Dortmund, Germany
`{jan.jurjens,holger.schmidt}@cs.tu-dortmund.de`

³ Fraunhofer Institut für Software- und Systemtechnik, Germany

⁴ Institut für technische Systeme GmbH, Germany

Abstract. Developing security-critical systems in a way that makes sure that the developed systems actually enforce the desired security requirements is difficult, as can be seen by many security vulnerabilities arising in practice on a regular basis. Part of the difficulty is the transition from the security requirements analysis to the design, which is highly non-trivial and error-prone, leaving the risk of introducing vulnerabilities. Unfortunately, existing approaches bridging this gap largely only provide informal guidelines for the transition from security requirements to secure design.

We present a *method* to systematically develop structural and behavioral UMLsec design models based on security requirements. Each step of our method is supported by *model generation rules* expressed as pre- and postconditions using the formal specification language OCL. Moreover, we present a concept for a *CASE tool* based on the model generation rules. Thus, applying our method to generate UMLsec design models supported by this tool and based on previously captured and analyzed security requirements becomes systematic, less error-prone, and a more routine engineering activity.

We illustrate our method by the example of a patient monitoring system.

1 Introduction

When building *secure systems*, it is instrumental to take *security requirements* into account right from the beginning of the development process to reach the best possible match between the expressed requirements and the developed software product, and to eliminate any source of error as early as possible. Knowing that building secure systems is a highly sensitive process, it is important to accomplish the transition from security requirements to secure design *correctly*, i.e., without introducing vulnerabilities.

In fact, there already exist a number of approaches to security requirements analysis (see [3] for an overview) and secure design (e.g., [10, 9]). Although this can be considered a positive development, the different approaches are mostly not integrated with each other. In particular, existing approaches on bridging the gap between security requirements analysis and design only provide informal

guidelines for the transition from security requirements to design. Carrying out the transition manually according to these guidelines is highly non-trivial and error-prone, which leaves the risk of inadvertently introducing vulnerabilities. Ultimately, this would lead to the security requirements not being enforced in the system design (and later its implementation).

We present a method to systematically develop structural and behavioral design models based on security requirements. We use a security requirement analysis method [6, 13] inspired by Jackson [8] that uses the UML (Unified Modeling Language)¹ profile *UML4PF* [5] to capture, structure, and analyze security requirements. We extend this approach by a detailed procedure for developing *UMLsec* [9] design models from previously captured and analyzed security requirements. Our method is supported by *model generation rules* expressed as pre- and postconditions using the formal specification language *OCL* (Object Constraint Language)². We present a concept for a *CASE tool* based on the model generation rules. Since our rules are specified in a formal and analyzable way, the implementation of this tool can be checked automatically for correctness with respect to the model generation rules. Consequently, applying our method to generate UMLsec design models supported by our tool and based on previously captured and analyzed security requirements becomes systematic, less error-prone, and a more routine engineering activity. We illustrate our method by the example of a patient monitoring system.

The rest of the paper is organized as follows: Section 2 introduces our security requirements engineering approach. We give a brief introduction into UMLsec in Sect. 3, which we use in Sect. 4 to systematically develop UMLsec design models based on previously captured and analyzed security requirements. We consider related work in Sect. 5. In Sect. 6, we give a summary and directions for future research.

2 Environment Description and Security Requirements Analysis

We propose a requirements engineering approach inspired by Jackson [8]. We illustrate this approach using the example of a *patient monitoring system*, which displays the vital signs of patients to physicians and nurses, and controls an infusion flow according to previously configured rules. In this setting, the display data and the configuration rules are transmitted over an insecure wireless network. We use this case study as a running example throughout this paper.

Security requirements can only be guaranteed for a certain context. Therefore, it is important to describe the *environment*, since software (called *machine*) is built to improve something in its environment. A *context diagram* represents the environment in which the machine will operate. Figure 1 shows the context diagram of the *PatientMonitoringSystem* (PMS) case study in UML notation with stereotypes defined in the UML profile *UML4PF* [5]. This profile is available online via <http://swe.uni-due.de/en/research/tool/>. Stereotypes give a specific meaning to the elements of a UML diagram they are attached to, and they are represented by labels surrounded by double angle brackets.

¹ <http://www.omg.org/spec/UML/2.3/Superstructure/PDF/>

² <http://www.omg.org/docs/formal/06-05-01.pdf>

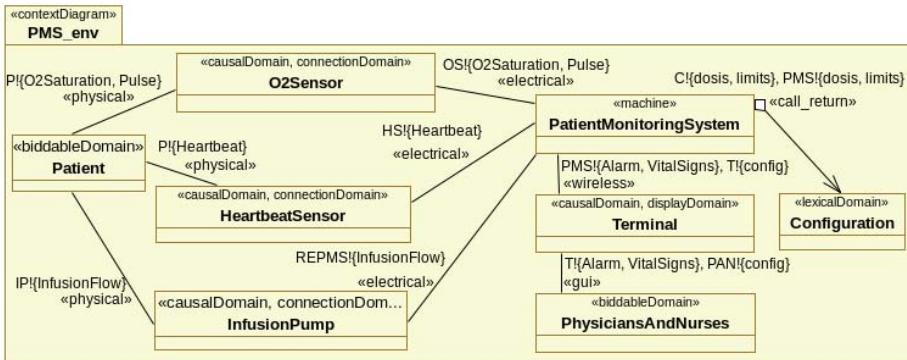


Fig. 1. Context Diagram of Patient Monitoring System

The machine is stereotyped **«machine»**, and in our example in Fig. 1 it is represented by the class **PatientMonitoringSystem**. A context diagram structures the environment using domains and interfaces. *Domains* describe entities in the environment. Jackson distinguishes the domain types *biddable domains* that are usually people, *causal domains* that comply with some physical laws, and *lexical domains* that are data representations. The domain types are modeled by the stereotypes **«BiddableDomain»** and **«CausalDomain»** being subclasses of the stereotype **«Domain»**. A lexical domain (**«LexicalDomain»**) is modeled as a special case of a causal domain. To describe the problem context in more detail, *connection domains* may be necessary. Connection domains establish a connection between other domains by means of technical devices. They are modeled as classes with the stereotype **«ConnectionDomain»**. Connection domains are, e.g., video cameras, sensors, or networks. A special type of connection domain is the *display domain* [2] for representing a display providing information. Display domains are modeled as classes with the stereotype **«DisplayDomain»**. The context diagram in Fig. 1 shows the biddable domains **Patient** and **PhysiciansAndNurses**, and the causal domains **O2Sensor**, **HeartbeatSensor**, **InfusionPump**, and **Terminal**. These causal domains are also connection domains, and the **Terminal** is a display domain.

Interfaces connect domains, and they contain *shared phenomena*. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain, as indicated by an exclamation mark. These interfaces are represented as associations, and the name of the associations contain the phenomena and the domain controlling the phenomena. For example, in Fig. 1 the notation **HS!{Heartbeat}** means that the phenomenon **Heartbeat** is controlled by the domain **HeartbeatSensor**.

Developers must elicit, examine, and describe the relevant properties of each domain. These descriptions form the *domain knowledge*. The domain knowledge consists of *assumptions* and *facts*. Assumptions are conditions that are needed, so that the *requirements* are accomplishable. Usually, they describe required user behavior. For example, it must be assumed that a user ensures not to be observed by a malicious user when entering a password. Facts describe fixed properties of the problem environment, regardless of how the machine is built.

Table 1. Functional Requirements of Patient Monitoring System

No	Requirement	«refersTo»	«constrains»
R1	The vital signs should be displayed, and an alarm should be raised if the vital signs exceed the limits.	Patient, Configuration	Terminal
R2	Physicians and nurses can change the configuration.	PhysiciansAndNurses	Configuration
R3	The infusion flow is controlled according to the configured doses for the current vital signs.	Patient, Configuration	InfusionPump

Table 2. Security Requirements of Patient Monitoring System

No	Security Statement	«complements»	«refersTo»	«constrains»/ Mechanism
1	Configuration should be protected from modification for Patient against Attacker or PhysiciansAndNurses should be informed.	R2	Configuration is asset, Terminal and WLAN know asset, Patient is stakeholder, against Attacker	Terminal-Display/ MAC of SSL
2	Alarm and Vital Signs should be protected from modification for Patient against Attacker or PhysiciansAndNurses should be informed.	R1	Alarm and Vital Signs are assets, Terminal and WLAN know asset, Patient is stakeholder, against Attacker	Terminal-Display/ MAC of SSL
3	Configuration, Alarm, and Vital Signs should be protected from disclosure for Patient against Attacker.	R1, R2	Configuration, Alarm, and Vital Signs are assets, Patient is stakeholder, against Attacker	WLAN/ encryption of SSL
4	The Shared Keys should be distributed to Terminal and PMS (for Patient) and Attacker should not be able to access Shared Keys.	R1, R2	Shared Keys are assets, Patient is stakeholder, against Attacker	WLAN/ key exchange of SSL (KE)

Domain knowledge and requirements are special statements. A statement is modeled as a class with a stereotype. In this stereotype, a unique identifier and the statement text are contained as stereotype attributes. When a requirement is stated, this means that something in the world should be changed by integrating the machine to be developed into it. Therefore, each requirement constrains at least one domain. This is expressed by a dependency from the requirement to a domain with the stereotype «constrains». A requirement may refer to several domains in the environment of the machine. For example, security requirements have to refer to an attacker of a certain strength. These references are expressed by a dependency from the requirement to a domain with the stereotype «refersTo». The domains referred are also given in the requirements description. Table 1 lists the functional requirements of the PMS case study.

Table 3. Security Domain Knowledge of Patient Monitoring System

No	Security Statement	«complements»	«refersTo»	«constrains»/ Mechanism
1	The KE keys should be distributed to Terminal and PMS for Patient, and Attacker should not be able to access Shared Keys.	R1, R2	KE keys are assets, Patient is stakeholder, against Attacker	WLAN/ manual import in physically protected area
2	Infusion Flow and PatientMonitoringSystem should be protected from modification for Patient against Attacker or Patient should know.	R1, R2, R3	Infusion Flow and Patient-Monitoring-System are assets, Patient is stakeholder, against Attacker	Infusion Pump, PatientMonitoring-System/ physical protection (e.g., EMF) and protection by Patient
3	Infusion Flow and PatientMonitoringSystem should be protected from disclosure for Patient against Attacker.	R1, R2, R3	Infusion Flow and Patient-Monitoring-System are assets, Patient is stakeholder, against Attacker	Infusion Pump, PatientMonitoring-System/ physical protection (e.g., EMF) and protection by Patient
4	Terminal should be protected from modification for Patient against Attacker or PhysiciansAndNurses should know.	R1, R2	Terminal is asset, Patient is stakeholder, against Attacker	Terminal/ physical protection (e.g., EMF) and protection by PhysiciansAndNurses
5	Terminal should be protected from disclosure for Patient against Attacker.	R1, R2	Terminal is asset, Patient is stakeholder, against Attacker	Terminal/ physical protection (e.g., EMF) and protection by PhysiciansAndNurses

Security requirements are associated with functional requirements, which we express using the stereotype «complements». For the functional requirements listed in Tab. 1, we initially identified some security requirements, as shown in Tab. 2 in rows 1-3, expressed as proposed in [5]. The required integrity (rows 1 and 2) supports the safety of the system and the required confidentiality (row 3) is necessary for privacy reasons. We decide on generic mechanisms that represent solutions of these requirements. To implement these mechanisms, additional domains have to be introduced, and additional requirements have to be fulfilled.

We choose the security mechanism MAC (Message Authentication Code) for integrity and symmetric encryption for confidentiality. For the mechanisms MAC and encryption, a Shared Key known by the Terminal and by the PMS is necessary. As required in Tab. 2 in row 4, this Shared Key must be distributed to the Terminal and to the PMS. The integrity and confidentiality of the Shared Key must be preserved. This will be implemented using a key exchange protocol. For the key exchange, additional secrets (KE keys) are necessary.

The KE keys should be distributed manually as described in Tab. 3 in row 1. Integrity and confidentiality of the Infusion Flow and the PatientMonitoringSystem should be ensured by physical protection (e.g., by reducing electromagnetic field (EMF) radiation and by protection against EMF radiation) and protection by Patient (e.g., Patient prevents physical access to the Infusion Flow) (Tab. 3 in rows 2 and 3). Integrity and confidentiality of the Terminal should be ensured by physical protection (e.g., by reducing electromagnetic field radiation and by protection against EMF radiation) and protection by PhysiciansAndNurses (Tab. 3 in rows 4 and 5).

For reasons of space, we do not depict the UML diagrams equipped with the mentioned stereotypes capturing these security requirements and the security domain knowledge. Instead, we present an overview of the security requirements and the security domain knowledge in Tabs. 2 and 3. These statements are the starting point for developing the design of the machine, which we achieve using UMLsec.

3 UMLsec

UMLsec constitutes a UML profile to develop and analyze security models. UMLsec offers new UML language elements, i.e., *stereotypes*, *tags*, and *constraints*, to specify typical security requirements such as secrecy, integrity, and authenticity, and attacker models. Examples for pre-defined UMLsec stereotypes are `<<critical>>` to label security-critical parts of UML diagrams, `<<secure dependency>>` to ensure that dependent parts of models preserve the security requirements relevant for the parts they depend on, `<<secure links>>` to introduce attacker models, and `<<data security>>` to analyze behavior models with respect to confidentiality and integrity requirements. The aforementioned stereotypes are used in the next section for creating UMLsec design models based on results from security requirements engineering. A detailed explanation and a formal foundation of the tags and stereotypes defined in UMLsec can be found in [9].

Based on UMLsec models and the semantics defined for the different UMLsec language elements, possible security vulnerabilities can be identified at a very early stage of software development. One can thus verify that the desired security requirements, if fulfilled, enforce a given security policy. This verification is supported by a tool suite, which is available online via <http://www.umlsec.de/>.

4 From Security Requirements to UMLsec Design Models

In this section, we connect the security requirements engineering approach presented in Sect. 2 with secure design based on UMLsec. We first present a procedure to generate UMLsec diagrams describing the environment in Sect. 4.1. Second, we introduce a procedure to generate UMLsec diagrams describing security mechanisms in Sect. 4.2. These procedures are supported by *model generation rules*, which we express using the formal specification language OCL. More precisely, the model generation rules consist of OCL *pre- and postconditions*. They can be considered as *patterns* that describe how existing security

measures and cryptographic protocols can be developed based on results from security requirements engineering.

We finally present in Sect. 4.3 work in progress on the construction of a tool that realizes the aforementioned procedures to develop UMLsec design models based on security requirements.

4.1 UMLsec Deployment Diagrams for Environment Descriptions

According to our security requirements engineering approach as illustrated in Sect. 2, describing the operational environment of a secure software system is of great importance. In fact, the environment description is also necessary for secure design: security-critical design decisions should lead to the fulfillment of the security requirements in the given environment. However, in a different environment, the same design decisions might lead to an insecure system.

In the following, we present a procedure to develop deployment diagrams enriched with UMLsec elements from context diagrams and security requirements. For each step, an operation name with parameters is provided. These operations represent model generation rules.

1. Create a UML package named adequately that contains a deployment diagram (*it is required that such a diagram does not yet exist and that exactly one context diagram exists*).

```
createDeploymentDiagram(diagramName: String)
```

2. Add the `<<secure links>>` stereotype to the package and assign a certain type of attacker (e.g., `default` or `insider` as described in [9, Chapter 4.1]) to the `{adversary}` tag. Decide which attacker type is appropriate based on threats modeled in the context diagram and domain knowledge collected during security requirements engineering. For example, `default` attackers cannot execute attacks in a LAN environment, but `insider` attackers can. Hence, if the context diagram describes an attack in a LAN environment, the attacker is of type `insider`.

```
addSecureLinksStereotype(inDiagram: String, adv: String)
```

3. Each domain contained in the context diagram (*it is required that exactly one context diagram exists and that the deployment diagram exists*) that is not a biddable domain is represented as a node in the deployment diagram.

```
createNodes(inDiagram: String)
```

4. Moreover, each domain that is part of another domain in the context diagram is represented either as a nested node or class.

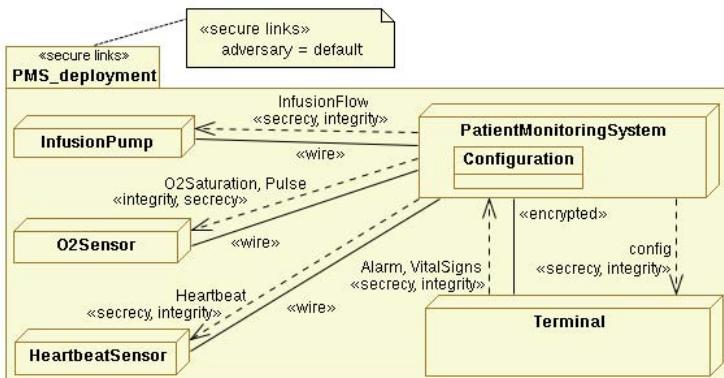
```
createNestedNodes(domainNames: String[]) or createNestedClasses( domainNames: String[])
```

5. Each connection between the aforementioned domains is represented as a communication path and a dependency:

- (a) We create a communication path stereotyped according to the communication type as described in Tab. 4. Note that only one of the UMLsec stereotypes is allowed for each communication path. Moreover, the defined mapping for context diagram stereotypes also applies to sub-stereotypes. For example, `<<wireless>>` is a sub-stereotype of `<<network_connection>>`, and therefore, `<<wireless>>` can be mapped to `<<Internet>>`, `<<LAN>>`, and `<<encrypted>>`, too.

Table 4. From Context Diagrams to UMLsec Deployment Diagrams

<i>Context Diagram</i>	<i>UMLsec Deployment Diagram</i>
«physical»	«wire» (physical protection against default adversary is assumed)
«ui»	not considered since biddable domains are not part of deployment diagrams
«remote_call»	see «network_connection»
«network_connection»	«Internet», «LAN», «encrypted» depending on the domain knowledge collected during security requirements engineering

**Fig. 2.** UMLsec Deployment Diagram Representing the Target State of Patient Monitoring System

We create communication paths for all relevant associations, and we also associate a communication type where no decision is necessary (`createCommunicationPaths(inDiagram: String)`). For all network connections (retrievable with `getNetworkConnections(): String[]`), the developer has to choose between **«Internet»**, **«LAN»**, or **«Encrypted»** (`setCommunicationPathType(inDiagram: String, assName: String, type: String)`).

- (b) We create a dependency stereotyped according to the control direction of the interfaces in the security requirement diagram and according to the following rules:

- The domain controlling the interface is translated into the target of the dependency.
- If more than one observing domains exist, the same number of dependencies must be introduced.
- If a confidentiality statement constraining the connection domain of the corresponding connection in the security requirement diagram exists, then the dependency is stereotyped **«secrecy»**.
- If an integrity statement referring to the connection domain of the corresponding connection in the security requirement diagram exists, then the dependency is stereotyped **«integrity»**.

`createDependencies(inDiagram: String)`

```

createDeploymentDiagram('PMS_Deployment');
addSecureLinksStereotype('PMS_Deployment', 'default');
createNodes('PMS_Deployment');
createNestedClasses({ 'Configuration'});
getNetworkConnections(); — returns {PMS!{ Alarm , VitalSigns }, T!{ config } }
createCommunicationPaths('PMS_Deployment');
setCommunicationPathType('PMS_Deployment', 'PMS!{ Alarm , VitalSigns },
T!{ config } ', encrypted');
createDependencies('PMS_Deployment');

```

Listing 1.1. Generating a UMLsec Deployment Diagram

```

1 createCommunicationPaths(inDiagram: String)
2 PRE Package.allInstances() ->select(name=diagramName)
3     ->size()=1 and
4     Package.allInstances() ->select(getAppliedStereotypes()
5         .name ->includes('ContextDiagram')) ->size()=1 and
6     Package.allInstances() ->select(getAppliedStereotypes()
7         .name ->includes('ContextDiagram')).clientDependency
8         .target ->select(oclIsTypeOf(Association)).oclAsType(Association)
9         ->select(not endType.getAppliedStereotypes().name
10             ->includes('BiddableDomain'))
11            .getAppliedStereotypes() ->forAll(rel_ass_st |
12                not rel_ass_st.name ->includes('ui') and
13                not rel_ass_st.general.name ->includes('ui') and
14                — similar for 'event', 'call_return', 'stream', 'shared_memory'
15            )
16 POST Package.allInstances() ->select(name=inDiagram).ownedElement
17     ->select(oclIsTypeOf(CommunicationPath))
18         .oclAsType(CommunicationPath)
19         .endType.name =
20             Package.allInstances() ->select(getAppliedStereotypes()
21                 .name ->includes('ContextDiagram')).clientDependency
22                 .target ->select(oclIsTypeOf(Association)).oclAsType(Association)
23                 ->select(not endType.getAppliedStereotypes().name
24                     ->includes('BiddableDomain')).endType.name and
25             Package.allInstances() ->select(getAppliedStereotypes()
26                 .name ->includes('ContextDiagram')).clientDependency
27                 .target ->select(oclIsTypeOf(Association)).oclAsType(Association)
28                 ->select(not endType.getAppliedStereotypes().name
29                     ->includes('BiddableDomain')) ->forAll(rel_ass |
30             Package.allInstances() ->select(name=inDiagram).ownedElement
31                 ->select(oclIsTypeOf(CommunicationPath))
32                     .oclAsType(CommunicationPath)
33                     ->exists(cp |
34                         cp.name = rel_ass.name and
35                         cp.endType.name = rel_ass.endType.name and
36                         ( cp.getAppliedStereotypes().name ->includes('physical') implies
37                             rel_ass.getAppliedStereotypes().name ->includes('wire')) and
38                         ( cp.getAppliedStereotypes().general.name ->includes('physical')
39                             implies
                                rel_ass.getAppliedStereotypes().name ->includes('wire'))
38
39     )

```

Listing 1.2. createCommunicationPaths(inDiagram: String)

The result of applying this method to the context diagram of the patient monitoring system shown in Fig. 1 is presented in Fig. 2. This UMLsec deployment diagram can be created following the command sequence depicted in Listing 1.1.

We now present the OCL specification of the model generation rule for step 5. Listing 1.2 contains the specification for step 5, generating the communication paths and stereotypes for those associations that can be derived directly. The first two formulas of the precondition of the model generation rule `createCommunicationPaths(inDiagram: String)` state that there does not exist a

package named equal to the parameter `diagramName` (lines 2-3 in Listing 1.2), and that there exists a package that contains a diagram stereotyped `<<Context-Diagram>>` (lines 4-5). The third formula of the precondition expresses that associations between transformed domains do not contain any of the `<<ui>>`, `<<event>>`, `<<call_return>>`, `<<stream>>`, `<<shared_memory>>`, stereotypes and subtypes (lines 6-15). If these conditions are fulfilled, then the postcondition can be guaranteed, i.e., names of nodes connected by each communication path are the same as the names of domains connected by an association in the context diagram (lines 16-29), and there exists for each relevant association contained in the context diagram a corresponding and equally named communication path in the deployment diagram that connects nodes with names equal to the names of the domains connected by the association. These communication paths are stereotyped `<<wire>>` if the corresponding associations are stereotyped `<<physical>>` or a subtype (lines 30-39).

4.2 UMLsec Class and Sequence Diagrams for Security Mechanism Descriptions

In the following, we show how to specify security mechanisms by developing UMLsec diagrams based on security requirements. For each communication path contained in the UMLsec deployment diagram developed as shown in Sect. 4.1 that is not stereotyped `<<wire>>`, we select an appropriate security mechanism according to the results of the problem analysis, e.g., MAC for integrity, symmetric encryption for security, and a protocol for key exchange, see Tab. 2). A security mechanism specification commonly consists of a structural and a behavioral description, which we specify based on the UMLsec `<<data security>>` stereotype. To create security mechanism specifications, we developed a number of model generation rules, for example:

- Securing data transmissions using MAC: `createMACSecuredTransmission(senderNodeName: String, receiverNodeName: String, newPackage: String)`
- Symmetrically encrypted data transmissions: `createSymmetricallyEncryptedTransmission(senderNodeName: String, receiverNodeName: String, newPackage: String)`
- Key exchange protocol: `createKeyExchangeProtocol(initiatorNodeName: String, responderNodeName: String, newPackage: String)`

Model generation rules can be regarded as *patterns* for security mechanism specifications. Each of the aforementioned model generation rules describes the construction of a package stereotyped `<<data security>>` containing structural and behavioral descriptions of the mechanism expressed as class and sequence diagrams. Moreover, the package contains a UMLsec deployment diagram developed as shown in Sect. 4.1.

We explain in detail the model generation rule `createKeyExchangeProtocol(initiatorNodeName: String, responderNodeName: String, newPackage: String)` shown in Listing 1.3. We use this protocol to realize the security requirement given in Table 2, row 4, of the patient monitoring system. We use the protocol that secures data transmissions using MACs for the security requirements in rows 1 and 2, and we use the protocol for symmetrically encrypted data transmissions for the security requirement in row 3.

```

1 createKeyExchangeProtocol(initiatorNodeName: String , responderNodeName:
2   String , newPackage: String);
PRE Node.allInstances() ->select(name=initiatorNodeName) ->size()=1 and
3   Node.allInstances() ->select(name=responderNodeName) ->size()=1 and
4   let cp-types: Bag(String) =
5     CommunicationPath.allInstances() ->select( cp |
6       cp.endType->includes(Node.allInstances()
7         ->select(name=initiatorNodeName)->asSequence()->first() )
8         and
9         cp.endType->includes(Node.allInstances()
10           ->select(name=responderNodeName)->asSequence()->first() )
11     ).getAppliedStereotypes().name
12   in
13     cp-types->includes('encrypted') or cp-types->includes('Internet ')
14     or cp-types->includes('LAN') and
15   Package.allInstances() ->select(name=newPackage) ->size()=0
16
17 POST Package.allInstances() ->select(name=newPackage) ->size()=1 and
18   ... Stereotype with attributes exists
19   Class.allInstances() ->select(name=initiatorNodeName)
20     ->select(oclIsTypeOf(Class)) ->size()=1 and
21   Class.allInstances() ->select(name=responderNodeName)
22     ->select(oclIsTypeOf(Class)) ->size()=1 and
23   ... dependencies with secrecy and integrity between initiator
24   and responder (both direction) created ...
25   Class.allInstances() ->select(name=initiatorNodeName)
26     ->select(oclIsTypeOf(Class)).ownedAttribute
27     ->select(name='inv(K_T)').type ->select(name = 'Keys') -> size()
28     = 1 and
29   ... other attributes exist...
30   Class.allInstances() ->select(name=initiatorNodeName)
31     ->select(oclIsTypeOf(Class)).ownedOperation
32     ->select(name='resp')
33     ->select( member->forAll(oclIsTypeOf(Parameter))) .member ->forAll(
34       par |
35       par->select( name->includes('shrd')) ->one(
36         oclAsType(Parameter).type.name->includes('Data')) xor
37         par->select( name->includes('cert')) ->one(
38           oclAsType(Parameter).type.name->includes('Data'))
39     ) and
40   ... other operations exist
41   ... stereotype and tags for initiator and responder class exist
42   let intera : Bag(Interaction) =
43     Package.allInstances() ->select(name=newPackage) .ownedElement
44     ->select(oclIsTypeOf(Collaboration))
45     .ownedElement ->select(oclIsTypeOf(Interaction))
46       .oclAsType(Interaction)
47   in
48     intera.ownedElement ->select(oclIsTypeOf(Lifeline))
49       .oclAsType(Lifeline).name ->includes(initiatorNodeName) and
50     intera.ownedElement ->select(oclIsTypeOf(Lifeline))
51       .oclAsType(Lifeline).name ->includes(responderNodeName) and
52     intera.ownedElement ->select(oclIsTypeOf(Message))
53       .oclAsType(Message).name
54       ->includes('init(N_i,K_T,Sign(inv(K_T),T::K_T))') and
55     intera.ownedElement ->select(oclIsTypeOf(Message))
56       .oclAsType(Message).name
57       ->includes('resp({Sign(inv(K_P_i),k_j::N)::K'_T},Sign(inv(K_CA),P_i::K_P_i))') and
58     intera.ownedElement ->select(oclIsTypeOf(Message))
59       .oclAsType(Message).name ->includes('xchd({s_i}..k)') and
60   ... conditions in sequence diagram exist

```

Listing 1.3. createKeyExchangeProtocol(initiatorNodeName: String, responderNodeName: String, newPackage: String)

The precondition of the model generation rule for key exchange protocols states that nodes named `initiatorNodeName` and `responderNodeName` exist (lines 2-3 in Listing 1.3). The communication path between these nodes (line 8)

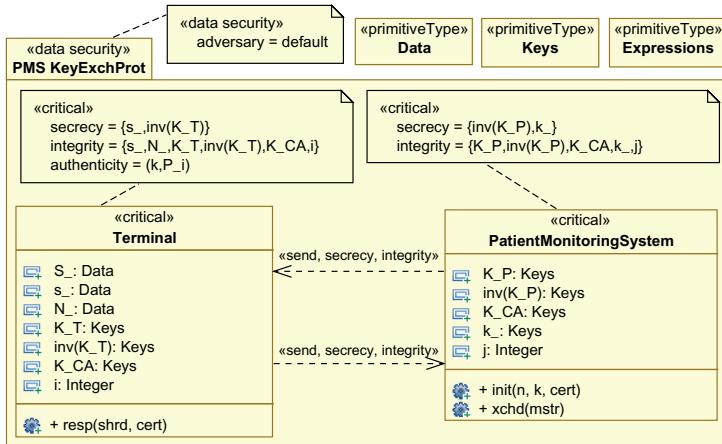


Fig. 3. Class Diagram of Key Exchange Protocol for Patient Monitoring System

should have the stereotype `<<encrypted>>`, `<<Internet>>`, or `<<LAN>>` (lines 9-10). Additionally, a package named `newPackage` must not exist (line 11). If these conditions are fulfilled, then the postcondition can be guaranteed. The first part of the postcondition describes the construction of a class diagram, and the second part specifies the construction of a sequence diagram. The following class diagram elements are created as shown in the example in Fig. 3:

- exactly one package named `newPackage` (line 13)
- stereotype `<<data security>>` and tags (`adversary`) for this package
- classes for initiator and responder named `initiatorNodeName` and `responderNodeName` (lines 15-16)
- dependencies with `<<secrecy>>` and `<<integrity>>` between initiator and responder (both directions)
- attributes for initiator and responder classes (lines 18-20)
- methods with parameters for initiator and responder class (lines 21-27)
- stereotype `<<critical>>` and corresponding tags (e.g., `secrecy`) for initiator and responder classes

The following sequence diagram elements are created as shown in the example in Fig. 4:

- lifelines for initiator and for responder in an interaction being part of a collaboration that is part of the created package (lines 29-34)
- messages in sequence diagram (lines 35-37)
- conditions in sequence diagram

A detailed description of this protocol pattern is given in [9, Chapter 5.2].

Figure 3 shows the class diagram and Fig. 4 the sequence diagram developed for the patient monitoring system according to this model generation rule. They are created with `createKeyExchangeProtocol('Terminal', 'PatientMonitoringSystem', 'KeyExchProt')`. In the created model, the tag `{secrecy}` of the `<<critical>>` class `Terminal` contains the secret `s_`, which represents an array of secrets to be exchanged in different rounds of this protocol. It also contains the

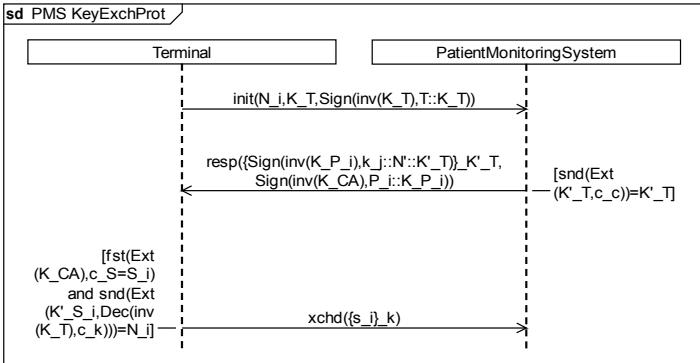


Fig. 4. Sequence Diagram of Key Exchange Protocol for Patient Monitoring System

private key $\text{inv}(K_T)$ of the Terminal. Next to these assets, the **{integrity}** tag additionally contains the nonces N_* used for the protocol, the public key K_T of the Terminal, the public key K_{CA} of the certification authority, and the round iterator i . These tag values are reasonable because the security domain knowledge in Tab. 3, rows 2 and 3 states that the PatientMonitoringSystem with its contained data is kept confidential and its integrity is preserved. The tag **{authenticity}** expresses that the PatientMonitoringSystem P_i is authenticated with respect to the Terminal. This is ensured by the domain knowledge in Tab. 3, row 1. The tag **{secrecy}** of the **<<critical>>** class PatientMonitoringSystem contains the session keys k_* and the private key $\text{inv}(K_P)$ of the PatientMonitoringSystem. The **{integrity}** tag consists of assets similar to the ones of the same tag of the Terminal. The tag **{authenticity}** is not used, since two-sided authentication is not necessary. Integrity and confidentiality of the data stored in the PatientMonitoringSystem (private key $\text{inv}(K_P)$, the public key K_P , the public key K_{CA} of the certification authority, and the round iterator j) is covered by the domain knowledge in Tab. 3, rows 4 and 5.

The sequence diagram in Fig. 4 specifies three messages and two guards, and it considers the i th protocol run of the Terminal, and the j th protocol run of the PatientMonitoringSystem. The sequence counters i and j are part of the Terminal and the PatientMonitoringSystem, respectively. The $\text{init}(\dots)$ message sent from the Terminal to the PatientMonitoringSystem initiates the protocol. If the guard at the lifeline of the PatientMonitoringSystem is true, i.e., the key K_T contained in the signature matches the one transmitted in the clear, then the PatientMonitoringSystem sends the message $\text{resp}(\dots)$ to the Terminal. If the guard at the lifeline of the Terminal is true, i.e., the certificate is actually for S and the correct nonce is returned, then the Terminal sends $\text{xchd}(\dots)$ to the PatientMonitoringSystem. If the protocol is executed successfully, i.e., the two guards are evaluated to true, then both parties share the secret s_i .

The key exchange protocol only fulfills the corresponding security requirements if integrity, confidentiality, and authenticity of the keys are ensured. According to our pattern system for security requirements engineering [5], applying the key exchange mechanism leads to dependent statements about integrity, confidentiality, and authenticity of the keys as stated in Tab. 3.

4.3 Tool Design

We are currently constructing a graphical wizard-based tool that supports a software engineer in interactively generating UMLsec design models. The tool will implement the model generation rules presented in the previous subsections to generate UMLsec deployment, class, and sequence diagrams. A graphical user interface allows users to choose the parameters, and it ensures that these parameters fulfill the preconditions. For example, users can choose the value of the second parameter of the model generation rule `setCommunicationPathType(inDiagram: String, assName: String, type: String)` based on the return values of the rule `getNetworkConnections()`. Our tool will automatically construct the corresponding parts of the UMLsec model as described in the postcondition. Since our model generation rules are specified with OCL in a formal and analyzable way, our tool implementation can be checked automatically for correctness with respect to our specification based on an appropriate API such as the Eclipse implementation for EMF-based models³. In addition to realizing the OCL specification, the tool will support workflows adequate to generate the desired UMLsec models, e.g., as depicted in Listing 1.1.

In summary, we presented in this section a novel integrated and formal approach connecting security requirements analysis and secure design.

5 Related Work

The approach presented in this paper can be compared on the one hand-side to other work bridging the gap between security requirements engineering secure design, and on the other hand-side to work on transforming UML models based on rules expressed in OCL.

Relatively little work has been done on the first category of related work, i.e., bridging the gap between security requirements analysis and design. Recently, an approach [12] to connect the security requirements analysis method *Secure Tropos* by Mouratidis et al. [4] and UMLsec [9] is published. A further approach [7] connects UMLsec with security requirements analysis based on heuristics. In contrast to our work, these approaches only provide informal guidelines for the transition from security requirements to design. Consequently, they do not allow to verify the correctness of this transition step.

The second category of related work considers the transformation of UML models based on *OCL transformation contracts* [1, 11]. We basically use parts of this work, e.g., the specification of transformation operations using OCL pre- and postconditions. Additionally, our model generation rules can be seen as patterns, since they describe the generation of completely new model elements according to generic security mechanisms, e.g., cryptographic keys.

6 Conclusions and Future Work

We presented in this paper a *novel method* to bridge the gap between security requirements analysis and secure design. We complemented our method by *formal model generation rules* expressed in OCL. Thus, the construction of UMLsec design models based on results from security requirements engineering becomes

³ Eclipse Modeling Framework (EMF):<http://www.eclipse.org/modeling/emf/>

more feasible, systematic, less error-prone, and a *more routine* engineering activity. We illustrated our approach using the sample development of a patient monitoring system.

In the future, we would like to elaborate more on the connection between the presented security requirements engineering approach and UMLsec. For example, we intend to develop a notion of correctness for the step from security requirements engineering to secure design based on the approach presented in this paper.

References

- [1] Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: OCL for the specification of model transformation contracts. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, Springer, Heidelberg (2004)
- [2] Côté, I., Hatebur, D., Heisel, M., Schmidt, H., Wentzlaff, I.: A systematic account of problem frames. In: Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP), pp. 749–767. Universitätsverlag Konstanz (2008)
- [3] Fabian, B., Gürses, S., Heisel, M., Santen, T., Schmidt, H.: A comparison of security requirements engineering methods. Requirements Engineering – Special Issue on Security Requirements Engineering 15(1), 7–40 (2010)
- [4] Giorgini, P., Mouratidis, H.: Secure tropos: A security-oriented extension of the tropos methodology. International Journal of Software Engineering and Knowledge Engineering 17(2), 285–309 (2007)
- [5] Hatebur, D., Heisel, M.: A UML profile for requirements analysis of dependable software. In: Schoitsch, E. (ed.) SAFECOMP 2010. LNCS, vol. 6351, pp. 317–331. Springer, Heidelberg (2010)
- [6] Hatebur, D., Heisel, M., Schmidt, H.: Analysis and component-based realization of security requirements. In: Proceedings of the International Conference on Availability, Reliability and Security (AReS), pp. 195–203. IEEE Computer Society Press, Los Alamitos (2008)
- [7] Houmb, S.H., Islam, S., Knauss, E., Jürjens, J., Schneider, K.: Eliciting security requirements and tracing them to design: An integration of common criteria, heuristics, and UMLsec. Requirements Engineering – Special Issue on Security Requirements Engineering 15(1), 63–93 (2010)
- [8] Jackson, M.: Problem Frames. Analyzing and structuring software development problems. Addison-Wesley, Reading (2001)
- [9] Jürjens, J.: Secure Systems Development with UML. Springer, Heidelberg (2005)
- [10] Lodderstedt, T., Basin, D.A., Doser, J.: SecureUML: A UML-based modeling language for model-driven security. In: Proceedings of the International Conference on the Unified Modeling Language (UML), London, UK, pp. 426–441. Springer, Heidelberg (2002)
- [11] Millan, T., Sabatier, L., Le Thi, T.-T., Bazex, P., Percebois, C.: An OCL extension for checking and transforming uml models. In: Proceedings of the WSEAS International Conference on Software Engineering, Parallel and distributed Systems (SEPADS), Stevens Point, Wisconsin, USA, pp. 144–149. World Scientific and Engineering Academy and Society (WSEAS), Singapore (2009)
- [12] Mouratidis, H., Jürjens, J.: From goal-driven security requirements engineering to secure design. International Journal of Intelligent Systems – Special issue on Goal-Driven Requirements Engineering 25(8), 813–840 (2010)
- [13] Schmidt, H.: A Pattern- and Component-Based Method to Develop Secure Software. Deutscher Wissenschafts-Verlag (DWV) Baden-Baden (April 2010)

Theoretical Aspects of Compositional Symbolic Execution

Dries Vanoverberghé* and Frank Piessens

Dries.Vanoverberghé, Frank.Piessens@cs.kuleuven.be

Abstract. Given a program and an assertion in that program, determining if the assertion can fail is one of the key applications of program analysis. Symbolic execution is a well-known technique for finding such assertion violations that can enjoy the following two interesting properties. First, symbolic execution can be *precise*: if it reports that an assertion can fail, then there is an execution of the program that will make the assertion fail. Second, it can be *progressing*: if there is an execution that makes the assertion fail, it will eventually be found. A symbolic execution algorithm that is both precise and progressing is a semi-decision procedure.

Recently, *compositional* symbolic execution has been proposed. It improves scalability by analyzing each execution path of each method only once. However, proving precision and progress is more challenging for these compositional algorithms. This paper investigates under what conditions a compositional algorithm is precise and progressing (and hence a semi-decision procedure).

Keywords: Compositional, symbolic execution, precision, progress.

1 Introduction

Given a program and an assertion in that program, determining whether the assertion can fail is one of the key applications of program analysis. There are two complementary approaches.

One can try to determine whether the assertion is *valid*, i.e. is satisfied in all executions of the program. This can be done using techniques such as type systems, abstract interpretation, or program verification. Such techniques are typically expected to be *sound*: if they report an assertion as valid, there will indeed be no execution that violates the assertion. However, these techniques suffer from false positives: they may fail to establish the validity of an assertion even if there is no execution that violates the assertion.

Alternatively one can look for counterexamples by trying to determine inputs to the program that will make the assertion fail. One important technique for this approach is symbolic execution [1], a well-known analysis technique to explore the execution traces of a program. The program is executed symbolically using logical symbols for program inputs, and at each conditional the reachability of both branches is checked

* Dries Vanoverberghé is a research assistant of the Fund for Scientific Research - Flanders (FWO).

using an SMT solver. When reaching the assertion, the analysis determines if it can find values for the symbolic inputs that falsify the assertion. Such a technique can not prove the validity of an assertion, but it has the advantage of avoiding false positives (a property that we will call *precision*). Obviously, sound and precise approaches are complementary. This paper focuses on precise algorithms, and more specifically on precise symbolic execution.

Thanks to many improvements to SMT solvers, symbolic execution has become an important technique, both in research prototypes [2,3,4,5,6,7,8,9,10] as well as in industrial strength tools [3,4]. Recently, *compositional* symbolic execution [11,12] attempts to further improve the scalability of symbolic execution. With compositional symbolic execution, each execution path of a method is only analyzed once. The results of this analysis are stored in a so-called *summary* of the method, and are reused by all callers of the method.

Traditional whole-program (non-compositional) symbolic execution has two interesting properties that are not necessarily maintained in the compositional case. First, as discussed above, symbolic execution can be *precise*: if it reports that an assertion can fail, then there is an execution of the program that will make the assertion fail. Proving precision for whole-program symbolic execution is relatively easy: one has to prove that symbolic execution correctly abstracts concrete executions, and that the SMT solver is sound and complete (which it can be for the class of constraints it needs to solve). Second, symbolic execution can *make progress* or be *progressing*: if there is an execution that makes the assertion fail, it will eventually be found. Therefore, there are no classes of programs where the analysis fails fundamentally. Again, making a symbolic execution algorithm progressing is relatively straightforward, for instance by making the algorithm explore the tree of possible paths through the program in a breadth-first manner. Since this tree is finitely-branching, a breadth-first exploration ensures that any node of the tree will eventually be visited. A symbolic execution algorithm that is both precise and progressing is a semi-decision procedure for the existence of counterexamples.¹

Although compositional symbolic execution is inspired by standard symbolic execution, the proofs of these important properties become much more challenging. In fact, some of the algorithms proposed recently are not necessarily semi-decision procedures. This paper develops proof techniques for showing precision and progress of compositional symbolic execution algorithms.

More specifically, this paper makes the following contributions:

- We formally model the existing compositional symbolic execution, based on a small but powerful programming language.
- We show that any compositional symbolic execution algorithm based on this formal model is *precise*.
- We give sufficient conditions for an algorithm to be *progressing*, and therefore be a semi-decision procedure.

¹ Note that precision is a soundness property, and progress is a completeness property, but we avoid the terms soundness and completeness on purpose to avoid confusion with soundness and completeness of verification algorithms or theorem provers.

For the purpose of investigating precision and progress, the assertion in the program is not relevant. What matters is whether the symbolic execution algorithm correctly enumerates all the reachable program states. Hence, for the rest of this paper, we will consider symbolic execution algorithms to be algorithms that enumerate reachable program states. Such an algorithm is precise if any program state that it enumerates is also reachable by the program. It is progressing if any program state reachable by the program is eventually enumerated.

The rest of this paper is structured as follows. First, in Section 2 we show by means of examples that precision and progress are hard to achieve for compositional algorithms. Then we introduce a small but powerful programming language in Section 3. Section 4 presents compositional symbolic execution and creates a formal model of it based on transition systems. Next, we show that this algorithm is precise and progressing (Section 5). Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Motivation

Traditional symbolic execution [1] explores paths through the program by case splitting whenever the execution reaches a branch. Since loops are also just branches that are encountered multiple times, this implies that loops are lazily unrolled, potentially an infinite amount of times². When a method call is reached, the target method is symbolically executed using the given arguments. Therefore, if the program calls a given method several times, the execution paths in that method will be re-analyzed for each call. The key idea of compositional algorithms is to avoid this repeated analysis. Instead, execution paths are explored for each method independently. The results of this exploration are stored in a *method summary*. Method calls are no longer inlined: a method call is analyzed in one single step and the result is computed based on the summary of the target method. Compositional symbolic execution has been shown [11,12,13] to improve performance, but maintaining precision and progress is challenging.

2.1 Precision

Compositional symbolic execution creates two potential causes of imprecision. First, when there is insufficient information about the calling context of a method, then one might conclude that unreachable program locations are reachable. For example, the highlighted statement in the method *P2* in Figure 1 is unreachable in the current program because the method *P1* only calls *P2* with argument $x \neq 0$. However, if one would analyze *P2* independently of *P1*, the analysis might conclude that the highlighted statement is reachable. In other words, since reachability is a whole-program property, we need to maintain some whole-program state even in a compositional analysis. The example algorithm we discuss later will do so by maintaining an invocation graph.

² Developer provided or automatically synthesized invariants can be used to create sound analyzers for particular classes of programs. This paper considers the general case where such invariants are not present or cannot be inferred.

Second, when a method returns and the analysis loses information about the relation between the arguments of the method and the return value, then the analysis might incorrectly conclude that a program location is reachable. For example, the highlighted statement in the method $P1$ in Figure 1 is unreachable. When the analysis over-approximates the result of $P2$ by the relation $result == 0 \vee result == 1 \vee result == -1$, then the highlighted location is reachable. To maintain precision, method summaries should not introduce such approximations.

```

int P1(int x) {
    if(x != 0){
        int r2 = P2(x);
        if(x > 0 && r2 != 1) return -1;
    }
    return 0;
}

int P2(int u) {
    if(u == 0) return 0;
    else if(u > 0) return 1;
    else return -1;
}

```

Fig. 1. Example program for precision

2.2 Progress

Non-compositional symbolic execution builds one global execution tree where leaf nodes represent either final program states, unreachable program states, or program states that require further analysis. Given a fair strategy to select such leaf nodes for further analysis, it is easy to show that the depth of the highest unexplored node keeps increasing and hence that any finite execution path will eventually be completely analyzed. This implies progress for non-compositional symbolic execution.

For compositional symbolic execution, the situation is more complex due to two reasons. First, as we discussed above, in order for method summaries to be precise, they must depend on the calling context. Hence, the discovery of a new call site for a method may increase the number of reachable points in the method and unreachable leaf nodes need to be reanalyzed taking into account the new calling context.

Secondly, when analyzing a method call, a compositional analysis relies on the summary of the target method for computing the return value. However, method summaries change over time when the analysis discovers new returns. As a consequence, nodes that were deemed unreachable based on the summary of the method must be reanalyzed when that method summary is updated.

The progress argument for non-compositional symbolic execution relies essentially on the fact that unreachable leaf nodes remain unreachable for the rest of the analysis. With compositional symbolic execution, this premise is no longer satisfied. Furthermore, it is impossible to guarantee that any finite execution path within the execution tree of a single method will eventually be completely analyzed. The program in Figure 2 provides an example of this phenomenon.

First, we explain the program: The two highlighted statements are both unreachable, and therefore the method $M1$ returns 0 for any input. To understand this, two invariants are important: First, the method $M1$ only calls the method $M2$ with parameters $u = v$ with $u > 0$. Second, if the parameters u and v of $M2$ are greater than zero, then $M2$ returns the minimum of u and v .

```

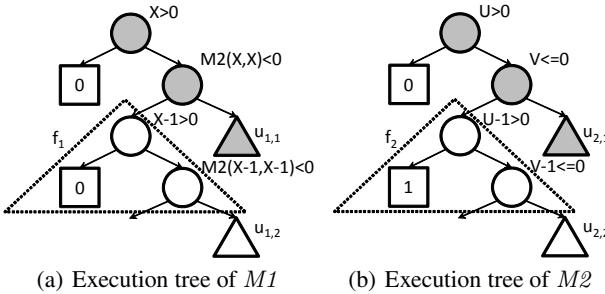
int M1(int x) {
    while (x > 0) {
        int y = M2(x, x);
        if(y < 0) return -1;
        x--;
    }
    return 0;
}

int M2(int u, int v) {
    int w = 0;
    while (u > 0) {
        if(v <= 0) return -w;
        u--; v--; w++;
    }
    return w;
}

```

Fig. 2. Example program for progress

Figures 3(a) and 3 show the execution trees of $M1$ and $M2$. Each circle represents a case split in the program, and the corresponding condition is written in the upper-right corner. From a circle, the arc to the left (right) means that the condition is false (true). Squares are final nodes, and imply that the method returns with the return value written inside the square. Triangles are unreachable nodes.

**Fig. 3.** Execution trees

Let $u_{i,j}$ be the analysis step that checks the j -th unreachable node of M_i , and f_i the sequence of analysis steps that explores the reachable part of the execution tree of M_i . The sequence f_1 causes a new invocation from $M1$ to $M2$ and therefore resets the unreachable nodes in $M2$. The sequence f_2 causes a new return and therefore resets the unreachable nodes in $M1$. Let $u_{i,2..}$ be the sequence $u_{i,2}, \dots, u_{i,n}$ where $u_{i,n}$ checks the deepest unreachable node of M_i . Suppose we analyze according to the fair schedule $f_1, f_2, [u_{2,1}, f_1, u_{2,2..}, u_{1,1}, f_2, u_{1,2..}]^*$. Then the highlighted nodes in the execution trees, that are only at depth 3 in the tree will be of status needs-further-analysis an infinite number of times. Hence, the depth of the analysis never stays larger than 3, and the given schedule is a counter example for the traditional proof of progress.

This shows that progress for compositional symbolic execution can not be proved by mimicking the proof for the non-compositional case on a per-method basis. In Section 5, we will propose an alternative technique to prove progress for compositional symbolic execution.

3 Programming Language

In this section, we introduce a small intermediate language that is particularly well-suited for presenting compositional symbolic execution. It only retains the structure

of the program that is essential: the structure of the control flow graph per procedure, and the calls and returns between procedures. The language focuses on sequential programs. Besides this restriction, all relevant more complicated language features can be translated to this core (e.g. parameters, return values or loops, . . .). For brevity, we also assume that the program does not contain (mutually) recursive methods. Although the algorithm in 4 depends on this simplification, our prototype implementation supports recursion by inferring ranks, but the details are beyond the scope of this paper.

A program p is a tuple $\langle M_p, G_p, m_p^0 \rangle$ where M_p is a set of methods, G_p is a set of global variables and $m_p^0 \in M_p$ is a distinguished entry method. Each method definition m for the program p is a tuple $\langle L_m, N_m, \lambda_m, n_m^0 \rangle$ where L_m is a finite set of local variables disjoint from the global variables G_p , N_m is a finite set of program locations, $n_m^0 \in N_m$ is a distinguished entry node and $\lambda_m : N_m \rightarrow Commands_{m,p}$ maps each node to a command. The sets of local variables and nodes of different methods are disjoint.

A command c for the method m of the program p is either:

- An assignment **assign** x, e, n where $x \in L_m \cup G_p$, e is a side-effect free expression over $L_m \cup G_p$ and constants, and $n \in N_m$ is a program location. This command updates the value of the variable x , and continues in location n .
- A conditional **if** e, n_t, n_f where e is a side-effect free expression over $L_m \cup G_p$, and $n_t, n_f \in N_m$ are program locations. If the expression e evaluates to true (false) the execution continues in location n_t (n_f).
- A call **call** m_t, n where $m_t \in M_p$ is the target method and $n \in N_m$ is a program location. This command invokes the method m_t and continues in location n .
- A return **ret** returns from the current method.

For each variable v , $\mathcal{D}(v)$ represents the value domain of the variable. A valuation σ_V is a partial function that maps each variables $v \in V$ to a value $val \in \mathcal{D}(v)$. Each domain has a default element $\mathcal{D}_0(v)$, and the default valuation σ_V^d for a set of variables V maps each variable $v \in V$ to $\mathcal{D}_0(v)$.

An execution state $s \in S_p$ for the program p is tuple $\langle \sigma_G, \bar{f} \rangle$ where

- σ_G is the current valuation for G_p
- $\bar{f} \in F_p^{*}$ is a sequence of frames for p (sequences are either empty (*nil*) or a concatenation $h; \bar{t}'$ of a head h and a tail \bar{t}')

A frame $f \in F_p$ for the program p is a tuple $\langle m, n_m, \sigma_{L_m} \rangle$ where

- $m \in M_p$ is the current method.
- $n \in N_m$ is the current program location.
- σ_{L_m} is the current valuation for L_m

The operational semantics $\rightarrow \subseteq S \times S$, gives an interpretation to the commands (More details can be found in a technical report [14]), and \rightarrow^* is its reflexive transitive closure. The execution of a program p starts in the initial state $s_p^0 = \langle \sigma_{G_p}^0, f_{m_p^0}^0 \rangle$ with $f_{m_p^0}^0$ the initial frame for m_p^0 , and $\sigma_{G_p}^0$ the input valuation for the global variables G_p . The initial frame for a method m is $f_m^0 = \langle m, n_m^0, \sigma_{L_m}^d \rangle$.

A state s is reachable from a state s' if and only if $s \rightarrow^* s'$. A state s is reachable in a program pr (denoted as $\models reach(pr, s)$) if and only if s is reachable from the initial state s_{pr}^0 of the program.

4 Compositional Symbolic Execution

In this section, we model the essence of existing compositional symbolic execution algorithms in order to formally study their precision and progress properties. Symbolic execution [1] is a technique to explore the execution paths of a program under all possible inputs. Instead of using a concrete input, the execution of the program is started with symbols representing arbitrary values. As a result, the values in the symbolic execution state are symbolic expressions that depend on the input symbols. Symbolic interpretation lifts the interpretations of commands to symbolic states.

For each execution path, symbolic execution constructs a *path condition*, a constraint in function of the input symbols that characterizes when an input follows that path. At a branch with condition C , the result of concrete execution is statically unknown: Either C is true and execution continues with the true-branch, or C is false and the false-branch is taken. Therefore, symbolic execution splits the set of inputs in two new sets: one where C is added to the path condition and one where the negation of C is added to the path condition. To check reachability, i.e. whether there is an execution that follows a path, a constraint solver checks satisfiability of the path condition. As a result, symbolic execution explores a prefix of the (potentially infinite) execution tree of the program. The resulting prefix is a partition of the global input space of the program.

The difference between compositional [11,12] and traditional symbolic execution is in the treatment of method calls and returns. In traditional symbolic execution, a call adds a new symbolic frame for the target method and continues execution until a return command pops this frame. Therefore, when a method is called twice, a path through the method is computed twice, even if it is guaranteed to follow the same path. Compositional symbolic execution explores the execution trees of each method in isolation. This results in a partition for each method (also called the summary). When a call is encountered, compositional symbolic execution uses the summary of the target method to compute the effect on the symbolic state.

To show that compositional symbolic execution is a semi-decision procedure, it is convenient to model the algorithm as a transition system $a \Rightarrow a'$ (and \Rightarrow^* its reflexive transitive closure) which starts in an initial analysis state a^0 . Non-terminating runs of the algorithm can be truncated after any number of transitions. In addition, the predicate $\vdash^a \text{reach}(p, s)$ denotes that the analysis concludes the reachability of the state s in the program p in an analysis state a .

Such a transition system is precise if and only if the conclusion in any reachable analysis state is sound³:

Definition 1 (Precision). For each program pr , concrete state s , and analysis state a such that $a_{pr}^0 \Rightarrow^* a$, $\vdash^a \text{reach}(pr, s)$ implies $\models \text{reach}(pr, s)$.

Obviously, compositional symbolic execution is not complete⁴ in any reachable analysis state and due to undecidability this is even impossible. However, the analysis incrementally discovers more and more reachable states. This incremental nature is captured by monotonicity:

³ Sound as a bugfinder, i.e. any state which is concluded reachable is truly reachable.

⁴ Complete as a bugfinder.

Definition 2 (Monotonicity). For each program pr , concrete state s , and analysis states a, a' such that $a \Rightarrow a'$, if $\vdash^a \text{reach}(pr, s)$ then $\vdash^{a'} \text{reach}(pr, s)$.

For a monotonous analysis, progress is the next best thing with respect to completeness: for any reachable concrete state, eventually there is an analysis state that concludes reachability for that concrete state:

Definition 3 (Progress). For each program pr and each concrete state s , if $\models \text{reach}(pr, s)$ then for all analysis states a' such that $a_{pr}^0 \Rightarrow^* a'$ there exists an analysis state a such that $a' \Rightarrow^* a$ and $\vdash^a \text{reach}(pr, s)$. In other words, for each reachable concrete state s , there always eventually is an analysis state that concludes s is reachable.

When an analysis is precise, monotonous and progressing, it is a semi-decision procedure.

4.1 Overview

The analysis state maintains a summary per method, which is a set of leaf nodes of the current prefix of the execution tree of the method. A leaf node $\langle stat, \nu, pc \rangle$ contains:

- A status $stat$, which is either unknown, finished or unreachable,
- A symbolic execution state ν ,
- A path condition pc .

The path condition defines the inputs (i.e. the values of the global variables) that will drive the execution of the method along this path. The symbolic execution state represents the state of execution after executing the path. The status indicates whether (a) the path is a complete path through the method, i.e. the method returns after this path (finished status) (b) the path is unreachable (unreachable status) (c) further exploration of continuations of this path are needed (unknown status). Symbolic execution states are defined like concrete execution states, except that all valuations are symbol valuations i.e. any variable has a symbolic expression instead of a concrete value.

The summaries only maintain per-method information. As we have shown in Section 2, it is necessary to maintain some whole program information in order to be precise. In particular, it is important to precisely track reachable method invocations and returns.

Initially, only the main method is reachable. As the analysis progresses, any call that is discovered is stored in an invocation graph. This graph is represented as a set of invocations, where each invocation is a tuple $\langle m_s, m_t, \varsigma_G, pc \rangle$. The methods m_s and m_t are the source and target methods, and ς_G and pc are the symbolic values of the global variables and the path condition at the moment of the invocation. Reachability checking will use the information in the call graph to decide whole-program reachability.

To support discovery of new returns efficiently, the return values of method calls can be modeled using logic function symbols[12]. The symbolic execution of a call is defined in terms of these function symbols. The interpretations of the function symbols are constructed using the current summary. As the analysis progresses, they become precise for more and more inputs. We discuss this in more detail in Section 4.2.

In addition, the analysis tracks all reachable program states it has enumerated. For this purpose, the analysis state contains a set of leaf nodes that succeeded the reachability check for each method. Based on this information, reachability conclusion is defined. If a leaf node $\langle stat, \nu, pc \rangle$ is in the reachable set of the method m in a given analysis state a , then any concretization of its symbolic state ν with global variables satisfying pc is concluded reachable in m . A state s is concluded reachable in an analysis state a (denoted $\vdash^a \text{reach}(pr, s)$) if and only if either

- s is concluded reachable in the entry method m_{pr}^0 in a or
- there is a state s' such that $\vdash^a \text{reach}(pr, s')$ and s' calls m and s is reachable in m .

Usually, one is only interested whether a point in a program is reachable (e.g. a location n in a method m). Therefore, implementations often store reachable program points instead of leaf nodes or avoid the reachability set completely by reporting an error when reaching a distinguished error-location. However, reachability conclusion of arbitrary states is essential for inductive invariants that enable the precision and progress proofs.

Finally, an analysis state can be defined as a tuple $\langle sum, invs, rs \rangle$ where

- sum is a function that maps each method m to a set of leaf nodes (its summary).
- $invs$ is a set of invocations.
- rs is a function that maps each method to a set of reachable leaf nodes.

In the initial analysis state a_p^0 , the invocation graph and the sets of reachable leaf nodes are empty. Each summary starts with a symbolic execution state at the entry of the method, where the value of all global variables contains a new symbol.

The high level overview of one step of the compositional symbolic execution algorithm is shown in Figure 4. During each step, the algorithm chooses a method m and an leaf node $\epsilon \in sum_a(m)$ with unknown status. Then, the algorithm checks whether there exists an input $\sigma_{G_p}^0$ such that the execution enters the method m and the global variables satisfy the path condition pc_ϵ of ϵ ($\text{Check}(a, m, \epsilon.pc)$). If there is no such input, the status of the ϵ is changed to unreachable. Otherwise, ϵ is added to the set of reachable leaf nodes of m , and symbolic execution continues with the interpretation ($SyInt(a, m, \epsilon)$) of the symbolic state ν_ϵ of ϵ . When symbolic interpretation finishes, it returns a set of new equivalence leaf nodes, and the current leaf node ϵ is replaced by the new leaf nodes ($ReplaceLeaf$). All method calls in this algorithm are guaranteed to terminate, and therefore one step of the algorithm always terminates.

We now zoom in on some aspects of the algorithm that are of importance for precision and progress.

4.2 Symbolic Interpretation

In this section, we informally discuss the inference rules for symbolic interpretation ($SyInt$). Full details are included in a technical report [14].

As pointed out in the previous section, the analysis uses uninterpreted function symbols to support discovery of new returns as the analysis progresses. The algorithm models the effect of the method m on the global variable v as an uninterpreted function symbol $rv_{m,v}$. When a method m is called with global variables ς_{G_p} , then the function application $rv_{m,v}(\varsigma_{G_p})$ models the value for the global variable v after executing m .

```

AnalysisState Step(AnalysisState a) {
     $(m, \epsilon) = Choose(a);$ 
    if(Check(a, m, \epsilon.pc)) {
         $a' = AddReachable(a, m, \epsilon);$ 
         $(a'', \pi) = SyInt(a', m, \epsilon);$ 
        return ReplaceLeaf(a'', m, \epsilon, \pi);
    } else {
        return MarkUnreach(a, m, \epsilon);
    }
}

```

Fig. 4. High level algorithm of symbolic execution

In addition, the method summaries are *partial*: there is no information about unexplored paths through a method. To deal with this, the algorithm models the set of global variable valuations that follow a finished path as an uninterpreted predicate rc_m .

During reachability checking, the algorithm computes the interpretation for the uninterpreted symbols using the method summaries (Figure 5) and replaces them using substitution.

$$\begin{aligned}
interps(sum) &= \bigcup_{m \in M_p} interp(m, \{\epsilon | \epsilon \in sum(m), stat_\epsilon = fin\}) \\
interp(m, \pi) &= rc_m \mapsto interp_{rc}(m, \pi) \cup (\bigcup_{v \in G_p} rv_{m,v} \mapsto interp_{rv}(m, v, \pi)) \\
interp_{rc}(m, \pi) &= \bigvee_{\langle fin, \nu, pc \rangle \in \pi} pc \\
interp_{rv}(m, v, \emptyset) &= \mathcal{D}_0(v) \\
interp_{rv}(m, v, \langle fin, \nu, pc \rangle \cup \pi) &= ite pc \leq_{G_\nu(v)} interp_{rv}(m, v, \pi)
\end{aligned}$$

Fig. 5. Interpretation of uninterpreted function symbols

For precision, it is essential that the interpretation of the uninterpreted symbols is precise: Whenever the return condition $rc_m(\sigma_{G_p})$ is true, the execution of the method m starting with global variables σ_{G_p} eventually reaches a return command, and each global variable v must equal $rv_{m,v}(\sigma_{G_p})$.

The treatment of assignment and branches is similar to the treatment in non-compositional symbolic execution: For an assignment, symbolic interpretation performs the same operation but on symbolic expressions instead of concrete values. For branches, symbolic interpretation creates a new leaf node for each branch and conjoins the branch condition or its negation to the path condition.

The rule call creates a new leaf node where the return condition is added to the path condition, and the return values are used to update the global variables. As mentioned in Section 2, some leaf nodes can become reachable by performing a call, and progress requires that all such leaf nodes are reconsidered. The algorithm conservatively reconsiders all unreachable leaf nodes of methods that are transitively reachable in the invocation graph by marking them as unknown (using the function $rec(a, m)$, defined more precisely in Appendix). In practice, more intelligent re-evaluation strategies can take the context into account in order to minimize the number of affected nodes, but this is beyond the scope of the formal model.

The return rule marks the unknown leaf node as finished, and thereby the interpretations of the current method change. In addition, the return rule marks all unreachable leaf nodes that depend on the return condition as unknown again (using the function $rer(a, m)$, also defined in Appendix). This is again essential to maintain progress.

For precision, the symbolic interpretation algorithm must maintain precision of the leaf nodes, i.e. if an input is a member of a leaf node, then the execution starting with that input eventually reaches the concretization of the symbolic state (the symbolic state after replacing the input symbols with the concrete input). In addition, all invocations $\langle m_s, m_t, \varsigma_G, pc \rangle$ in the invocation graph must be precise: If an input satisfies the condition pc , then the execution of m_s starting with that input reaches a call to the method m_t and the global variables are the concretization of ς_G .

For progress, it is essential that symbolic interpretation maintains *totality* of the summaries. A reachable concrete state s is on the frontier if all predecessors in the execution to s are concluded reachable, but s is not concluded reachable. The summaries are total if any concrete state s on the frontier is a concretization of some unknown leaf node ϵ in the summary of some method m . Informally, this is a kind of completeness guarantee for symbolic interpretation. For any concrete state on the frontier, the analysis can make the “right” choice. Totality implies that leaf nodes may not be marked unreachable if *Check* succeeds in the current analysis state. For this reason, the call and return rules need to reconsider some unreachable leaf nodes.

4.3 Reachability Checking

Finally, to check reachability ($Check(a, m, pc)$), the algorithm globalizes the path condition pc based on the invocation graph inv_a , substitutes the symbols $\varsigma_{G_p}^0$ by their interpretation $interps(sum_a)$, and uses an SMT-solver to check the satisfiability of the resulting constraints. The globalization $glob_p(a, m, pc)$ globalizes the constraint pc in the context of m using the invocation graph inv_{s_a} and is defined inductively as follows:

- If $m = m_p^0$ then $glob_p(a, m, pc) = pc$
- If $m \neq m_p^0$ then $glob_p(a, m, pc) =$
 $\bigvee_{\langle m_s, m, \varsigma_G, pc' \rangle \in inv_{s_a}} glob_p(a, m_s, pc' \wedge pc[\bigcup_{v \in G_p} \varsigma_{G_p}^0(v) \mapsto \varsigma_G(v)])$

In the absence of recursion, the invocation graph is cycle free, and the inductive definition is well-founded.

For precision, it is important that $Check(a, m, pc)$ only returns true when there is a reachable state s where the execution enters m and the global variables satisfy pc (*precision of Check*). This follows from precision of the leaf nodes, the precision of the interpretations and the soundness of the SMT-solver as a satisfiability checker.

The contrary is not the case: If there is an execution that enters m where the global variables satisfy pc , $Check(a, m, pc)$ need not return true because this execution might follow an unexplored path through some method. For progress, it is only necessary that $Check(a, m, pc)$ holds if the execution that enters m where the global variables satisfy pc only uses concrete states that are concluded reachable (*Restricted completeness*). This requires completeness of the SMT-solver as a satisfiability checker.

4.4 Implementation

To show that our framework captures the essence of compositional symbolic execution, we have implemented one instantiation of the framework for the intermediate language of the .NET platform [15]. For bytecode manipulation, we use the Mono.Cecil [16] library and as constraint solver we use Z3 [17]. Our implementation achieves similar speedups as other compositional symbolic execution tools [12,11].

5 Properties

In this section, we show that the algorithm of Section 4 is precise, and we show that it is also progressing as long as the choices are fair. We only give a rough outline of the proof, since a more detailed exposition does not fit in the page limits. More details can be found in a technical report [14].

First, we show that compositional symbolic execution is precise.

Theorem 1 (Precision). *The algorithm is precise.*

Proof. To show precision, it suffices to show by induction over \Rightarrow^* that the following properties are satisfied for each reachable analysis state a :

- Each leaf node ϵ of the summary $sum_a(m)$ of each method m is precise.
- Each finished leaf node ϵ of the summary $sum_a(m)$ of each method m is returning from m .
- All invocations $inv \in invs_a$ are precise, and the invocation graph is cycle free.
- The interpretations are precise.
- $Check(a, m, pc)$ is precise for each method m and condition pc .
- Each leaf node ϵ in rs_a is precise, and $Check(a, m, pc_\epsilon)$ succeeds (where m is active method of the symbolic state of ϵ).

Then, we show that compositional symbolic execution is monotonous.

Theorem 2 (Monotonicity). *For each program pr , concrete state s , and analysis states a, a' such that $a \Rightarrow a'$, if $\vdash^a \text{reach}(pr, s)$ then $\vdash^{a'} \text{reach}(pr, s)$.*

Proof. Follows from the fact that (a) \Rightarrow never removes reachable leaf nodes ($rs_a \subseteq rs_{a'}$). (b) \Rightarrow never removes invocations ($invs_a \subseteq invs_{a'}$).

Since the search tree of the algorithm is potentially infinite, monotonicity is not sufficient to find all reachable states: The algorithm might get stuck exploring only a subspace of the program. Fortunately, this can not happen if the analysis is *fair*, i.e. if each unknown node is eventually chosen by the algorithm.

Definition 4 (Fairness). *An application strategy of the compositional symbolic execution algorithm is fair if and only if for any analysis state a such that $a_{pr}^0 \Rightarrow^* a$, for any unknown node $\epsilon \in sum_a(m)$, the algorithm always eventually chooses $\langle m, \epsilon \rangle$.*

Next, the progress argument relies on the validity of the following properties in each reachable analysis state a (which can again be proven by induction over \Rightarrow^*):

- The summaries sum_a are total.
- $Check(a, m, pc)$ is restricted complete for any method m and constraint pc .

Finally, we show that compositional symbolic execution algorithm is progressing if it is fair. The proof shows a slightly stronger property, namely that there always eventually is an analysis state where all concrete states on the execution trace that reaches s are concluded reachable. This is essential since it gives a stronger induction hypothesis: we assume that all but the last concrete state s is concluded reachable and we show that the analysis always eventually reaches an analysis state where s is also concluded reachable. This hypothesis is necessary since a state might only be reachable from one invocation that has not yet been discovered, whereas its predecessor is already reachable based on another invocation. Together with totality of the summaries and restricted completeness of reachability checking, this allows a compact and intuitive proof for progress.

Theorem 3 (Progress). *If the compositional symbolic execution algorithm is fair, then it is progressing.*

Proof. By induction on \rightarrow^* .

Base step. If s is the initial state, then $\vdash^a \text{reach}(pr, s)$ holds after applying the only possible analysis step.

Induction step. If $s_{pr}^0 \rightarrow^* s'$ and $s' \rightarrow^* s$ and there always eventually is a reachable analysis state a' such that all concrete states from s_{pr}^0 to s' are concluded reachable in a' , then we must show that there always eventually is a reachable analysis state a such that $\vdash^a \text{reach}(pr, s)$. If $\vdash^{a'} \text{reach}(pr, s)$ already holds, then the proof is trivial.

1. First, we show that there exists an unknown node $\epsilon \in sum_{a'}(m)$ such that $Check(a', m, pc_\epsilon) = \text{true}$ and s is a concretization of ϵ . This means that if we choose $\langle m, \epsilon \rangle$, then the state s will become reachable in the next analysis state. This follows from the fact that the summaries are total, and restricted completeness of check.
2. By fairness, there always eventually exists a reachable analysis state a'' such that $\langle m, \epsilon \rangle$ has not been chosen yet, and is chosen in the next analysis step. Since $\langle m, \epsilon \rangle$ has not been chosen, it must still be in the summary of m ($\epsilon \in sum_{a''}(m)$). Because invocations are never removed ($invs_a \subseteq invs_{a''}$), the method $Check$ is monotonous and $Check(a'', m, pc_\epsilon) = \text{true}$. Therefore, if $\langle m, \epsilon \rangle$ is chosen in $a'' \Rightarrow a$, then $\vdash^a \text{reach}(pr, s)$.

6 Related Work

Compositional symbolic execution was first introduced in the context of SMART [11], as an extension of the automatic testing tool DART [18]. The authors informally argue that SMART is sound and complete (as a bugfinder) relatively to DART. In addition, DART is always sound (precise) and it is complete when it terminates [18]. The precision proofs depends critically on the dynamic aspect of SMART and DART. This paper

only depends on the precision of the interpretation rules. When the interpretation rules are imprecise in SMART or DART, it either causes incompleteness or non-termination. In addition, the progress property is stronger than completeness upon termination.

With demand-driven compositional symbolic execution [12], the dependency on the inner-most first search order of SMART is lifted. To achieve this, function summaries are encoded in the SMT-solver. In addition, the algorithm allows the SMT solver to construct inputs that follow unexplored paths through some methods. Such inputs may not reach their actual target but they always explore some new part of the program. This may be useful to alleviate the imperfectness of SMT solvers. We did not incorporate this in our framework, but the results can easily be extended. The authors claim relative completeness (as a bugfinder), and termination for programs with finite amounts of paths. The progress property in this paper is less algorithm specific and therefore more clear. In addition, it lifts the need for a termination argument. In the absence of fairness, demand-driven compositional symbolic execution does not satisfy the stronger progress property.

Finally, the system SMASH [13] combines the aspect of compositional analysis with may-must alternation. SMASH significantly outperforms both may-only, must-only and non-compositional may-must analysis. The analysis in this paper is a must analysis. As part of the soundness argument, the authors show that the must analysis of SMASH is precise. In addition, they show that the may analysis of SMASH is sound. Unfortunately, the combination of a sound may analysis with a precise must analysis is not necessarily a semi-decision procedure.

7 Conclusion

This paper creates a formal framework for compositional symbolic execution, based on a small but powerful calculus. We have modeled compositional symbolic execution as a transition system and formalized the meaning of precision and progress. In addition, we have proven that the algorithm is precise, and makes progress if the choices are fair. Finally, we have shown preliminary results of an implementation of the algorithm that is precise and progressing, and hence is a semi-decision procedure.

References

1. King, J.C.: Symbolic execution and program testing. *Commun. ACM* 19(7), 385–394 (1976)
2. Tillmann, N., de Halleux, J.: Pex-white box test generation for.net. In: Proc. of Tests and Proofs 2008, pp. 134–153. Springer, Berlin (2008)
3. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: automatically generating inputs of death. In: Proc. of CCS 2006, pp. 322–335 (2006)
4. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: NDSS. The Internet Society, SanDiego (2008)
5. Nori, A.V., Rajamani, S.K., Tetali, S., Thakur, A.V.: The YOGI project: Software property checking via static analysis and testing. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 178–181. Springer, Heidelberg (2009)
6. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: end-to-end containment of internet worms. *SIGOPS Oper. Syst. Rev.* 39(5), 133–147 (2005)

7. Brumley, D., Hartwig, C., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Song, D.: BitScope: Automatically dissecting malicious binaries. Technical Report CS-07-133, School of Computer Science, Carnegie Mellon University (March 2007)
8. Anand, S., Pasareanu, C.S., Visser, W.: JPf-SE: A symbolic execution extension to Java Pathfinder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 134–138. Springer, Heidelberg (2007)
9. Molnar, D.A., Wagner, D.: Catchconv: Symbolic execution and run-time type inference for integer conversion errors. Technical Report 2007-23, University of California Berkeley (February 2007)
10. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: Proc. of SIGSOFT 2008/FSE-16 (2008)
11. Godefroid, P.: Compositional dynamic test generation. In: Proc. of POPL 2007, pp. 47–54 (2007)
12. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 367–381. Springer, Heidelberg (2008)
13. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.D.: Compositional may-must program analysis: unleashing the power of alternation. SIGPLAN Not. 45(1), 43–56 (2010)
14. Vanoverberghe, D., Piessens, F.: Precise and progressing compositional symbolic execution: Extended version (2010),
<http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW582.abs.html>
15. European Computer Machinery Association: Standard ECMA-335: Common Language Infrastructure. 4th edn. (June 2006)
16. Evain, J.: Cecil, <http://www.mono-project.com/Cecil>
17. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008),
http://dx.doi.org/10.1007/978-3-540-78800-3_24
18. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. SIGPLAN Not. 40(6), 213–223 (2005)

Testing Container Classes: Random or Systematic?

Rohan Sharma¹, Milos Gligoric¹, Andrea Arcuri²,
Gordon Fraser³, and Darko Marinov¹

¹ University of Illinois, Urbana, IL-61801, USA

{sharma27, gliga, marinov}@illinois.edu

² Simula Research Laboratory, Lysaker-P.O. Box 134, Norway

arcuri@simula.no

³ Saarland University, Saarbruecken-66123, Germany

fraser@cs.uni-saarland.de

Abstract. Container classes such as lists, sets, or maps are elementary data structures common to many programming languages. Since they are a part of standard libraries, they are important to test, which led to research on advanced testing techniques targeting such containers and research on comparing testing techniques using such containers. However, these techniques have not been thoroughly compared to simpler techniques such as random testing. We present the results of a larger case study in which we compare random testing with shape abstraction, a systematic technique that showed the best results in a previous study. Our experiments show that random testing is about as effective as shape abstraction for testing these containers, which raises the question whether containers are well suited as a benchmark for comparing advanced testing techniques.

1 Introduction

Automation of test generation is an important and still open issue, regularly leading to new techniques and refinements of existing techniques. The empirical evidence on research in this area often focuses on container classes [4, 5, 6, 7, 10, 12, 14, 17, 19, 22, 23, 26, 27, 29, 30, 31, 33, 34]—containers are an important part of many standard libraries, and bugs in these containers could significantly affect applications, so directing testing efforts to these containers is worthwhile. Testing containers is not only important but also challenging to achieve with some advanced testing technique such as those based on symbolic execution [30].

Automating testing for containers is convenient because they usually do not interact with the environment and can be tested without construction of complex input data [21]. Any results achieved on containers for one language easily carry over to other languages, as the data structures are generic and implemented for many different languages. However, precisely these aspects of containers also mean that simple techniques such as random testing could be able to achieve good results. Unfortunately, the literature offers little evidence on how more advanced techniques compare to random testing. In fact, excluding comparisons with search algorithms, we are aware of only one study that compared random testing to systematic techniques, by Visser, Pasareanu, and Pelanek [30]; we will refer to this study as the *VPP* study.

The VPP study proposed several advanced techniques for test input generation for Java container classes and compared these techniques against one another and with random testing on four container classes. The comparison metrics were basic block coverage and a simplified version of *predicate coverage* [11] that measures how many combinations of program predicates are covered (which differs from the traditional condition or MCDC coverages [3]). The results showed that among the advanced techniques the best was *shape abstraction* (described in detail in Section 4.2). The results in the VPP study also showed that shape abstraction was the same as random testing for basic block coverage but even better than random testing for predicate coverage.

In this paper, we perform a larger set of experiments to compare random testing and shape abstraction, which remains the best technique in systematic test generation for containers. Our study substantially extends on VPP in several important aspects:

Number of containers: We use a total of *13 different container classes* in our evaluation, four from VPP and nine more that were used previously in various other studies [17, 25].

Types of containers: We consider containers implemented with both *pointer-based*, *linked structures* and *array-based structures*, whereas VPP (and several other studies) used only containers implemented with pointer-based structures.

Metrics: We use *mutation score* [3], in addition to predicate coverage, for comparison of techniques. To the best of our knowledge, this is the first study that relates predicate coverage and mutation scores. We also measure predicate coverage more thoroughly than the VPP study which considered only a few manually selected program points whereas we use a semi automated tool to consider all branches. To make it easier for other researchers to experiment with predicate coverage, we made our instrumented code publicly available at <http://mir.cs.illinois.edu/coverage>.

Statistical Analysis: We perform a rigorous *statistical analysis* of the results, as opposed to VPP which had no statistical analysis.

Results: The experiments show that although random testing is much faster than shape abstraction, *random testing still achieved comparable predicate coverage and mutation scores to those achieved by shape abstraction*. Specifically, random testing was better for four containers, shape abstraction was better for five containers, and the results were inconclusive for four containers. In contrast, the VPP study found shape abstraction better than or equal to random testing for all four containers considered. Our experiments also raise the concern that containers should not be used as a *de facto* benchmark for comparing advanced testing techniques because random testing can work very well for containers.

Bugs: While the goal of our study was to compare random testing and shape abstraction but not necessarily look for bugs, we still *found three real bugs* in two containers used in previous studies [17, 25]. All three bugs were found by random testing, were missed by the advanced techniques used in previous studies, and were confirmed by the original authors of the respective container code.

One relevant aspect in which our study evaluates less than the VPP study is that we test only two basic methods/operations for each container (namely, *add* and *remove*),

Table 1. Sample recent papers that use containers among subjects in the case studies

Authors	Year	Reference	#Subjects	#Containers	%Containers
Tonella	2004	[27]	6	5	83%
Visser <i>et al.</i>	2004	[29]	1	1	100%
Xie <i>et al.</i>	2004	[33]	11	9	81%
Xie <i>et al.</i>	2005	[34]	7	7	100%
Visser <i>et al.</i>	2006	[30]	4	4	100%
Wappler and Wegener	2006	[31]	4	4	100%
d'Amorim <i>et al.</i>	2006	[14]	16	12	75%
Inkumsah and Xie	2008	[19]	13	10	77%
Arcuri and Yao	2008	[10]	7	7	100%
Andrews <i>et al.</i>	2008	[4]	2	1	50%
Arcuri	2009	[6]	1	1	100%
Ribeiro <i>et al.</i>	2009	[22]	2	2	100%
Ribeiro <i>et al.</i>	2010	[23]	2	2	100%
Baresi <i>et al.</i>	2010	[12]	15	9	60%
Arcuri	2010	[7]	6	6	100%
Andrews <i>et al.</i>	2010	[5]	34	34	100%
Staats and Pasareanu	2010	[26]	6	4	66%
Galeotti <i>et al.</i>	2010	[17]	6	6	100%

whereas the VPP study tested a larger number of methods/operations for two of their four containers.

2 Related Work

A number of studies compared advanced techniques for test generation or test selection with random testing [14, 15, 16, 18, 32], but these studies did not provide conclusive answers either way (sometimes random testing looked better and sometimes worse than more advanced techniques), and they did not focus on containers. Several recent techniques use random generation for object-oriented unit tests [1, 13, 21] but target shallower exploration of larger codebases and can generate complex test data inputs, while testing containers focuses on deeper exploration of smaller codebases and typically requires only simple data inputs. In this paper we focus on test generation for containers.

While clearly not all testing studies use only containers for evaluation [21], containers are still widely used in many recent studies on testing. Table 1 shows a sample of 18 papers that either propose techniques specifically for testing containers or use containers as subject code to evaluate new or existing testing techniques. As can be seen, containers are a large percentage of subjects used for evaluations, even when the number of subjects is not very high. Our evaluation uses 13 containers. While several of these studies evaluate effectiveness of random testing in various scenarios [5, 6, 7, 10, 14, 30, 33], only the VPP study [30] directly compares random testing and advanced systematic techniques (not based on search). In terms of metrics used, branch coverage is the most

```

public class TreeSet {
    int size;
    TreeSetEntry root;

    public TreeSet() { ... }
    public boolean add(int aKey) { ... }
    public boolean remove(int aKey) { ... }
    ...
}

class TreeSetEntry {
    int key;
    boolean color;
    TreeSetEntry left;
    TreeSetEntry right;
    TreeSetEntry parent;
}

```

(a) Parts of the TreeSet class

```

public class TreeSetTest {
    public void test1() { // length 1
        TreeSet s = new TreeSet();
        s.add(5);
    }
    public void test2() { // length 1
        TreeSet s = new TreeSet();
        s.remove(5);
    }
    public void test3() { // length 2
        TreeSet s = new TreeSet();
        s.add(5);
        s.remove(21);
    }
    ...
}

```

(b) Example tests for TreeSet

Fig. 1. This is a typical example of a container class, where testing focuses on a few selected methods (*add* and *remove* in this case). A test case starts with the default constructor for the container, which creates an empty instance. Then, on this container the *add* and *remove* methods are repeatedly called. The length of the test case is the number of such calls.

represented. The only exceptions are predicate coverage used in the VPP study [30], statement coverage [5], MCDC [26], and an unspecified “structural coverage” [22, 23]. We use not only predicate coverage, which subsumes branch coverage, but also mutation score. Only a few of these studies use statistical analyses [5, 6, 7, 10]. We also present a statistical analysis of our experimental results.

3 Example

We next describe an example that illustrates the problem of test generation for containers. Figure 1(a) shows partial code for the `TreeSet` container class that we obtained from a previous study by Galeotti *et al.* [17]. This class implements a set of integer values using red-black trees. Each `TreeSet` object has a number of nodes and a pointer to the root node. Each `TreeSetEntry` node stores a value, a color (which can be red or black), and pointers to the left and right children and a parent. The methods for the `TreeSet` class include those to create the empty set, add an element to the set, and remove an element from the set.

Figure 1(b) shows an example of automatically generated tests for the `TreeSet` class. Each test creates an empty set and has a sequence of add and remove operations. The tests are written in the JUnit format [2], but note that these tests have no assertions, i.e., they do not assert that the methods should return certain values. The assumption in this automated generation of test inputs is that outside oracles are used to validate the execution; in the simplest case, one can use a generic oracle that requires each test to terminate regularly, i.e., without throwing an uncaught exception. The goal of generation, hence, is to produce tests that achieve high coverage for some testing criteria.

While the goal of our experiments was to compare the coverage obtained by random testing and shape abstraction, we still found some bugs. For example, using random testing, we found two bugs in the `TreeSet` code from [17]. These bugs resulted in `NullPointerExceptions`. Note that they were missed by the advanced testing techniques [17] because these techniques did not generate appropriate test inputs.

4 Test Generation for Container Classes

Our aim is to provide more empirical evidence on how random testing compares to shape abstraction in the context of testing containers. To this extent, we generate test suites with the goal to maximize predicate coverage, and also use mutation score for comparison of techniques.

A test suite S consists of n test cases, $S = \{t_1, \dots, t_n\}$. In general there are different ways to represent and encode a test case. Because we focus on container classes, we use a simple representation that is common in the literature (e.g., [30]): A test case is a sequence of operations such as *add* and *remove* on a container instance created with its default constructor. For the input data, we only consider integer values bounded in $[1, R]$, where R is a fixed constant. The *length* $l(t)$ of a test case t is the number of operations. We do not consider the default constructor in the length. For a test suite S , we define its length as $l(S) = \sum_{t \in S} l(t)$.

4.1 Random Testing

Random testing (RT) is a fast testing technique, in which test cases are simply sampled at random from the input domain. Although RT is often considered a naive testing strategy [20], it can be very effective in many testing scenarios [9, 15]. When the test cases have a variable length representation, there can be different ways to sample test cases at random [9]. However, in this paper we fix the length and number of test cases in each sampled test suite.

Based on the problem definition from Section 4, we analyze the following strategy to generate test suites S . First fix a number n of test cases for S and generate n test cases t with a fixed length $l(t) = k$. The generated test suite S will have length $l(S) = n \times l(t) = nk$. In a random test case, each operation is uniformly chosen (i.e., *add* or *remove*), and the input data is uniformly chosen in $[1, R]$, where R is a constant.

Generating and running a small test suite of n test cases is quite fast for container classes. When the goal is to maximize predicate coverage, an option would be to run RT z times and then output the test suite with highest coverage out of the z runs. How to choose z ? This depends on the available testing budget (i.e., for how long a software tester is willing to wait to obtain test data). We can consider two options: (1) run RT for a predefined number of runs z , or (2) run RT several times and stop it after some amount of time (e.g., one second). In practical contexts, option (2) would be preferable and easier to apply. However, option (1) is easier to apply in empirical analyses, because it does not have to deal with the actual execution time (e.g., side effects of implementation/code details, unpredictable delays due to other processes running in parallel, etc.). In this paper, we use only option (1), although we still report indicative times to give a better picture of the techniques' performance.

```

1 // inputs: container C, length limit L, values bound R
2 void SA0 {
3     Queue<MethodSequence> ToExplore = empty_queue;
4     ToExplore.enqueue(empty_sequence);
5     Set<AbstractState> Explored = empty_set;
6     for (int i = 1; i <= L; i++) {
7         Queue<MethodSequence> NextToExplore = empty_queue;
8         foreach (MethodSequence s: ToExplore) {
9             for (Operation op: {"add", "remove", ...}) {
10                 int[] p = randomPermutationOfRange(1, R);
11                 for (int v: p) {
12                     MethodSequence s' = append(s, op(v));
13                     Container c = create empty C and execute sequence s';
14                     if (execution covered a new predicate combination)
15                         print(s'); // a new test is generated
16                     AbstractState a = abstract(c);
17                     if (a ∉ Explored) {
18                         Explored = Explored ∪ {a};
19                         NextToExplore.enqueue(s'); } } }
20     }
21     ToExplore = NextToExplore; }
22 }
```

Fig. 2. Pseudo-code for shape abstraction (SA) exploration

Once we obtain a test suite of length nk , many method calls might be redundant. Manually verifying the behavior of each operation (e.g., writing assert statements) would likely be too tedious/difficult if no automated oracle is available. Therefore, an approach to deal with this problem is to *minimize* the output test suite S generated by RT, but with the constraint of maintaining the same coverage of the original test suite. We use the following simple minimization algorithm [7]: Remove one method call at a time and re-execute the test case; if the coverage decreases, then re-introduce that method call in the test case. Given a total of nk method calls, this minimization algorithm would require the execution of nk test cases. However, in cases in which we want to make fair comparisons against other techniques, we might want the total length to be at least m function calls. When we minimize a test suite, we can simply stop once the size has reached m .

4.2 Shape Abstraction

The VPP study introduced (explicit execution with abstract matching based on) *shape abstraction* (SA) as a technique for test generation of containers. Unlike RT that produces random sequences of method calls, SA attempts to find that certain sequences are equivalent and hence need not be generated. The original exposition of SA [30] was based on explicit-state model checking, and SA was one of six techniques in the same general framework. We provide a new exposition that directly describes the exploration, focuses solely on SA, and allowed us to obtain a faster implementation of SA without relying on a model checker.

Figure 2 shows the pseudo-code for SA. It takes as input the container code with operations (such as *add* and *remove*), the maximum length of sequences of the

operations, and the bounds for the values for those operations. It produces as the output tests (i.e., method sequences) whose execution increases predicate coverage. SA performs a breadth-first search (up to length L) with randomized choices of values (from 1 to R). Line 10 randomly permutes the values to be explored. SA maintains a queue `ToExplore` of method sequences that still need to be explored and a set `Explored` of *abstract states* that were already encountered.

The key novelty of SA was to compute abstract states using shape abstraction, i.e., ignoring the concrete values in the containers and taking into account only the *shape* in which the container nodes are connected. For example, two red-black `TreeSet` objects that have the same shape of nodes (i.e., the same underlying connection starting from the `root` node and following the `left` and `right` pointers) would map into the same abstract state even if they had different values in those nodes. As a concrete example, consider two balanced trees that each have three nodes and the same red-black colors, one tree with the values 2 in the root, 1 in the left child, and 3 in the right child, and the other tree with the values 4 in the root, 2 in the left child, and 6 in the right child. SA would map these two trees into the same abstract shape.

SA starts the exploration with a queue that has only the empty sequence and with the empty set of abstract states. For each sequence s in the queue (line 8), it randomly chooses an operation and value to apply (lines 9 and 10), extends the sequence to s' , executes this sequence¹, prints the sequence if it covered some new predicate combination (lines 14 and 15), and checks if the exploration encountered a new abstract state that should be explored in the future (line 17). Notice that the sequence s' is included in the output test suite whenever its execution increases predicate coverage, even if s' results in an abstract shape that has been already explored and thus s' will not be extended.

5 Case Study

5.1 Subject Containers

Table 2 shows some basic statistics for the 13 subject containers used in our study. For each subject we list a brief identifier, the reference from which we directly obtained the source code, the number of lines of code, the number of mutants generated by the Javalanche mutation tool, and the parameter values for shape abstraction. While we obtained the code directly from three studies [17, 25, 30], all the containers were used previously in many other studies and were originally taken from various sources including Java libraries, textbook implementations done by students, and open source. We included some examples of different implementations of the same containers to see if there are differences in the results.

5.2 Predicate Coverage

Following the VPP study [30], our experiments use a simplified version of the predicate coverage testing criterion. The full predicate coverage, proposed by Ball [11],

¹ The original exposition in VPP [30] assumed a stateful model checker whereas we present SA based on re-execution of method sequences, which does not allow reusing a container from the previous exploration as it may have been modified.

Table 2. Statistics of the subject containers used in our evaluation. For shape abstraction, we set the same value for the length of sequence (L) and the bound for method values (R).

Container	Id	Reference	LOC	Mutants	$L = R$
AvlTree	C1	[17]	160	335	20
BinomialHeap	C2	[30]	225	289	33
BinTree	C3	[30]	94	126	13
FibHeap	C4	[30]	245	285	13
FibonacciHeap	C5	[25]	319	295	15
HeapArray	C6	[25]	75	122	25
IntAVLTreeMap	C7	[25]	160	199	20
IntRedBlackTree	C8	[25]	228	279	22
LinkedList	C9	[17]	176	335	3
NodeCachingLinkedList	C10	[17]	172	159	6
SinglyLinkedList	C11	[17]	76	167	5
TreeMap	C12	[30]	404	651	21
TreeSet	C13	[17]	248	360	22

is a strong criterion that measures how many combinations of *all* program predicates are covered at *all* program points. The predicates are taken from conditional statements and program assertions. For TreeSet, for example, the predicates include `t == null`, `aKey == t.key`, `t.left != null`, and many others. Unlike the traditional branch, condition, or MCDC coverages [3] that consider values of predicates only *near* where they are used in the code, predicate coverage considers values of predicates at all program points, including *far* from where they are used in the code. Predicate coverage requires using proper variables in scope; for instance, the `remove` method has a variable `TreeSetEntry p`, and predicate coverage would evaluate `p.left != null` (and all other predicates) although there is no such condition in that method.

To make the measurement tractable, the VPP study used only *some* program predicates, and we follow the same approach. However, unlike the VPP study that evaluated predicate coverage at *some* manually selected branches, we use semi-automated instrumentation to evaluate predicate coverage at *all* branches. Our instrumentation is not fully automatic as we manually select the variables for predicates. Describing the predicates and variables we used would be hard, so to enable comparative studies, we made our instrumented code publicly available at <http://mir.cs.illinois.edu/coverage>.

5.3 Mutation Analysis

Mutation analysis [3] is the process of systematically seeding syntactic changes into a program to determine whether the test cases can detect the resulting semantic program mutants. Undetected (“live”) mutants can guide the tester in improving a test suite, while detected (“killed”) mutants are used to quantify the effectiveness of a test suite in terms of its *mutation score* that is calculated as the ratio of killed mutants to all mutants.

With appropriate mutation operators, mutation analysis subsumes several traditional coverage criteria such as branch coverage [3]. We are not aware of any study on

relationship of mutation analysis and predicate coverage. But an important difference to code coverage is that mutation analysis does not simply check whether some piece of the code has been executed: To (strongly) kill a mutant means to propagate the infected state to an observable output.

We consider output as follows. Given a test case that calls methods on an instance of a container class, we record the state of the container after execution on the original program, and compare it with the state of the container after execution on a mutant program. If there are observable differences in the state, this mutant is considered killed by the test case.

We have implemented this mechanism as an extension to the Javalanche [24] mutation system: Each test case is instrumented automatically with additional instructions that record and compare the state of a container at the end of a test case. To compare states with each other, we simply use the `toString` method, which is commonly overridden by the container classes. In addition, we make sure that all potential instance-specific substrings (e.g., @ followed by a hexadecimal number) are removed from this output to prevent false positives.

5.4 Experimental Design

For each of the 13 containers, we compare random testing (RT) against shape abstraction (SA). We first run SA, and as shown in Figure 2, it takes two parameters: L is the length of method sequence, and R is the bound for method values. Following the VPP study, we set $L = R$, and we choose the smallest value for L such that (1) the predicate coverage remains constant across 10 different random seeds for L and (2) this predicate coverage is the same for 10 seeds for $L - 1$. The values for L and R are shown in Table 2. We then run the SA experiments for 100 seeds with these bounds.

We run RT as follows. We use 2,000 iterations. For each iteration, a test suite of size $n = 5$ is generated, where each test case has length $k = 200$, so the total length of a test suite is equal to 1,000. Integer inputs are randomly chosen in $[1, R]$, where $R = 20$. After these 2,000 iterations, the test suite with maximum predicate coverage is selected. If several test suites have the same maximum coverage, one test suite is selected at random. This test suite is then minimized, setting the lower bound m for its total length to be the average (across 100 seeds) test suite length obtained with SA on the same container class. (This is the reason why we run SA first.) With successful minimization, this implies that the resulting test suite lengths will be, on average, about the same for both RT and SA. (Note that, in theory, a minimization could even increase predicate coverage while reducing the length of the test case/suite.) Because RT is affected by chance, to obtain enough data to reach reliable conclusions, for each of the 13 containers we ran RT for 100 seeds.

Notice that running RT for more than 2,000 iterations would likely lead to better results because we select from all the iterations one test suite with the highest predicate coverage. For example, we could run RT for the same amount of time that SA takes. However, the problem with doing that would have been the fairness of the comparisons. If two testing techniques (such as RT and SA) are run for the same amount of time, then the worse quality (e.g., measured with predicate coverage) of one technique could be just due to some inefficiencies in the technique’s implementation. If a technique has

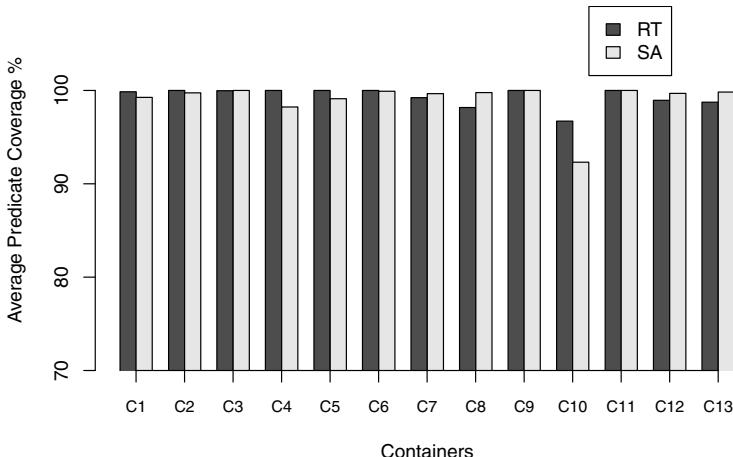


Fig. 3. Average predicate coverage for both random testing (RT) and shape abstraction (SA)

better quality, to increase confidence in the validity of such results, the technique should be also faster. On our machines, running RT for 2,000 iterations takes on average a few seconds, whereas SA is roughly *seven times slower*. In other words, RT consumes less computational resources, and thus its better quality (if any) would have strong validity.

5.5 Results for Predicate Coverage and Mutation Score

Figure 3 shows, for each container, the average predicate coverage divided by the maximum coverage obtained for that container. Specifically, for each container, we first calculated the highest coverage M out of the 200 test suites (100 generated by RT and 100 generated by SA). Then, we divided by M the average predicate coverage for RT (100 test suites) and for SA (100 test suites). The reason for using M is twofold: (1) many predicate combinations could be simply infeasible, and we cannot know how many are feasible, and (2) the number of predicate combinations in various containers is very different, and plotting them without normalizing the data would have led to graphs that are difficult to compare.

Figure 4 presents the results for the mutation analysis, where the average number of killed mutants is reported for each container and testing technique. In contrast to the results for predicate coverage, we did not normalize the data for mutation score. The low mutation score for containers (C1, C3, C4, C9, C10, C11) is partly due to our test generation focusing only on the methods for *add* and *remove* operations, whereas many mutants of these containers are also contained in other methods; likely, extending test generation to include other methods would increase the mutation score. In addition, there is always a number of equivalent mutants which cannot be killed. Because detecting equivalent mutants is an undecidable problem, we included these equivalent mutants in the total number of mutants that was used to calculate the mutation scores.

To analyze these data by taking into account the random components of the techniques, we followed a rigorous statistical procedure [8]. For both comparisons based on

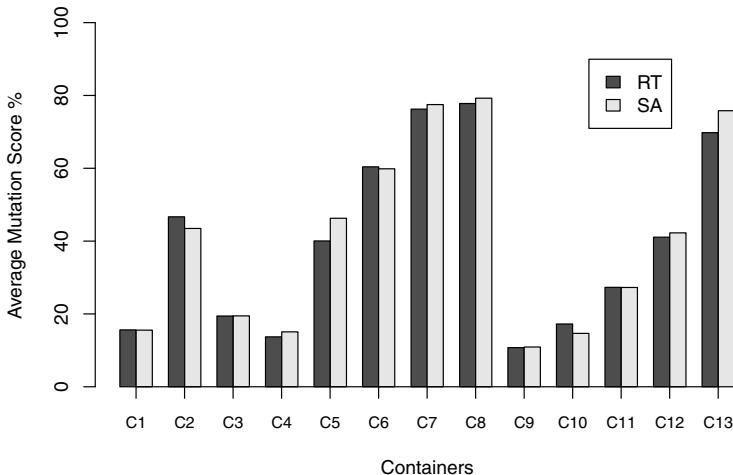


Fig. 4. Average mutation score for both random testing (RT) and shape abstraction (SA)

predicate coverage and based on mutation score, for each container, when we compare RT against SA, we used a Mann-Whitney U-test to assess whether the effectiveness of these two techniques is statistically different. The resulting *p-values* of these statistical tests indicate the probability of Type I error, i.e., the probability of wrongly stating that there is a difference in quality when actually there is no difference.

To assess the magnitude of the difference in a standardized way (i.e., the so called *effect size*), we use the Vargha-Delaney \hat{A}_{12} statistic [28] to compare the quality of RT against SA. In our context, this effect size is an estimate of the probability that a run of RT would give better result than a run of SA. If there is no difference, then we would expect $\hat{A}_{12} = 0.5$. On one hand, if we obtain $\hat{A}_{12} = 1$, this would mean that in *all* the 100 runs of RT we obtained better results than in *all* the 100 runs of SA. On the other hand, if $\hat{A}_{12} = 0$, then it would mean that SA was always better than RT. Table 3 reports the obtained p-values (for the Mann-Whitney U-test) and the \hat{A}_{12} measures (for the Vargha-Delaney statistic).

5.6 Random Testing vs. Shape Abstraction

The experiments show that RT and SA are about equally effective for these 13 containers and the two metrics. For predicate coverage, $\hat{A}_{12} > 0.5$ for five cases, $\hat{A}_{12} < 0.5$ for six cases, and $\hat{A}_{12} = 0.5$ for two cases. For mutation score, $\hat{A}_{12} > 0.5$ for six cases, and $\hat{A}_{12} < 0.5$ for seven cases. Considering the relative behavior of RT against SA, the results for predicate coverage are largely similar to those for mutation score. However, in three of the 13 cases the technique that gives higher predicate coverage does not also give higher mutation score.

Consider first C1. For predicate coverage, the difference is very small (\hat{A}_{12} close to 0.5), and the p-value is rather high, which we can interpret as RT and SA basically behaving similarly. For mutation score, however, the difference is still small (\hat{A}_{12} close to 0.5), but the p-value is rather low, which we can interpret as RT being better than SA for mutation score and thus for C1 overall.

Table 3. Results of the statistical analysis. The last column shows if random testing is better (RT), shape abstraction is better (SA), both are about equal (\approx), or the results are inconclusive (<>).

Container		Id	Predicate Coverage p-value	\hat{A}_{12} p-value	Mutation Score \hat{A}_{12}	Better Quality	
AvlTree		C1	0.512	0.487	0.059	0.564	RT
BinomialHeap		C2	0.001	0.555	0.001	1.000	RT
BinTree		C3	0.001	0.420	0.158	0.490	SA
FibHeap		C4	0.001	1.000	0.001	0.191	<>
FibonacciHeap		C5	0.001	1.000	0.001	0.005	<>
HeapArray		C6	0.013	0.530	0.001	0.821	RT
IntAVLTreeMap		C7	0.001	0.279	0.006	0.388	SA
IntRedBlackTree		C8	0.001	0.064	0.001	0.086	SA
LinkedList		C9	1.000	0.500	0.514	0.524	\approx
NodeCachingLinkedList		C10	0.000	0.785	0.001	0.937	RT
SinglyLinkedList		C11	1.000	0.500	0.322	0.505	\approx
TreeMap		C12	0.001	0.144	0.001	0.069	SA
TreeSet		C13	0.001	0.076	0.001	0.052	SA

Consider then C4 and C5. They are particularly interesting as RT is always better than SA for predicate coverage, but quite the opposite holds for mutation score. On average, SA achieves 1.38% and 6.24% higher mutation scores for C4 and C5, respectively. Looking at the difference in the sets of mutants killed by SA and RT for these two containers revealed that every single mutant killed by a SA test suite was also killed by at least one of the RT suites. This indicates that RT has a greater variance in mutation score even if it is relatively stable for predicate coverage, which is not surprising as our RT minimization focuses on predicate coverage and not all mutants are directly related to predicates.

Note that C4 and C5 are one example of different implementations of the same data structure. Recall that we intentionally included such implementations among our subjects to evaluate whether the differences between RT and SA depend on the details of the implementations. We find that they largely do not. For example, C4 and C5 behave the same way: RT is better for predicate coverage and SA for mutation coverage. All of C8, C12, and C13 are based on red-black trees, and for all three SA is better than RT (for both predicate coverage and mutation score). C9 and C11 are very similar list implementations, and RT and SA are approximately the same for both. In contrast, C10 is a more complex list implementation, and we find that RT is better than SA (which is consistent with the differences seen for C9 and C11, although those differences are too small to conclude that RT is better). Interestingly, for C1 and C7, which are both based on AVL balanced trees, RT is better than SA for C1, but SA is better than RT for C7.

Recall also that our subjects include not only pointer-based, linked structures (as in the VPP study) but also a container implemented with an array-based structure, namely C6. The results for C6 show that RT is clearly better than SA for this case, but we cannot generalize to all array-based structures.

To summarize, in the context of our study, we cannot identify a superiority of one of the two testing techniques with respect to either predicate coverage or mutation score.

However, there is indication that SA performs better for tree-like structures that require complex shapes for coverage (C3, C7, C8, C12, C13), whereas RT performs better for structures that require longer sequences for coverage (C1, C2, C6, C10).

5.7 Bugs

While the goal of our study was to compare RT and SA but not necessarily look for bugs, we still found three real bugs in two containers used in previous studies [17, 25]. Specifically, we found two bugs in the `TreeSet` code from TACO [17] and one bug in the `HeapArray` code from one of our previous studies [25]. The first two bugs led to `NullPointerExceptions`, while the third bug led to an infinite loop. We found all three bugs using RT, and all three bugs were missed by the advanced techniques used in previous studies because those techniques focused on more thorough testing with shorter tests and failed to generate longer tests necessary to reveal these bugs. We reported all three bugs to the original authors of the respective container code, and the authors confirmed them as real bugs and corrected them. The first two bugs were due to a copy-paste mistake, and the third bug was an error of omission.

We also found three bugs that we introduced by mistake in the testing infrastructure that exercised the container code. Specifically, we found one bug in `FibHeap` that resulted in an infinite loop because the test driver was removing a node that did not exist in the structure, and two bugs in `AvlTree` that resulted in `NullPointerExceptions` because our semi-automated instrumentation for measuring predicate coverage changed the original code. We corrected all these bugs, and all our experiments reported above were run with the corrected code.

6 Threats to Validity

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of having faults in our testing tools, we tested them and inspected surprising results. Furthermore, randomized algorithms are affected by chance. To cope with this problem, we ran each experiment 100 times, and we followed rigorous statistical procedures to evaluate their results.

Threats to *construct validity* are on how the quality of a testing technique is defined. We measured not only predicate coverage but also mutation score.

Threats to *external validity* regard the generalization to other types of software, which is common for any empirical analysis. However, in this paper we specifically target container classes and the implementation instances that are commonly used as a benchmark in the literature. The fact that random testing is very efficient in generating effective test cases for container classes will likely not hold for many other types of software. Note that shape abstraction does not apply to all types of software.

7 Conclusion

Containers are important and challenging to test, and many advanced testing techniques were developed for containers. However, there has not been much comparison of these

advanced testing techniques with simpler techniques such as random testing. We presented a larger case study that compared random testing with shape abstraction, a state-of-the-art systematic technique. Our experiments showed that random testing achieves comparable results as shape abstraction, but random testing uses much less computation resources than shape abstraction. We hope that our results provide motivation for future testing studies to (1) compare newly proposed advanced techniques to random testing and/or (2) evaluate newly proposed advanced techniques not (only) on containers but (also) on other code where random testing does not perform well.

Acknowledgments. We thank Marcelo Frias, Juan Pablo Galeotti, Corina Pasareanu, and Willem Visser for providing clarifications about their studies and code used in their experiments. We also thank David Schuler and Andreas Zeller for help with Javalanche. Andrea Arcuri is funded by the Norwegian Research Council. Gordon Fraser is funded by the Cluster of Excellence on Multimodal Computing and Interaction at Saarland University, Germany. This material is based upon work partially supported by the US National Science Foundation under Grant No. CCF-0746856.

References

1. Jtest, <http://www.parasoft.com/jsp/products/jtest.jsp>
2. JUnit, <http://junit.sourceforge.net/>
3. Ammann, P., Offutt, J.: *Introduction to Software Testing*. Cambridge University Press, Cambridge (2008)
4. Andrews, J.H., Groce, A., Weston, M., Xu, R.G.: Random test run length and effectiveness. In: International Conference on Automated Software Engineering (ASE), pp. 19–28 (2008)
5. Andrews, J.H., Menzies, T., Li, F.C.: Genetic algorithms for randomized unit testing. *IEEE Transactions on Software Engineering (TSE)* 99 (2010) (preprints)
6. Arcuri, A.: Insight knowledge in search based software testing. In: Genetic and Evolutionary Computation Conference (GECCO), pp. 1649–1656 (2009)
7. Arcuri, A.: Longer is better: On the role of test sequence length in software testing. In: International Conference on Software Testing, Verification and Validation (ICST), pp. 469–478 (2010)
8. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: International Conference on Software Engineering, ICSE (to appear, 2011)
9. Arcuri, A., Iqbal, M.Z., Briand, L.: Formal analysis of the effectiveness and predictability of random testing. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 219–229 (2010)
10. Arcuri, A., Yao, X.: Search based software testing of object-oriented containers. *Information Sciences* 178(15), 3075–3095 (2008)
11. Ball, T.: A theory of predicate-complete test coverage and generation. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 1–22. Springer, Heidelberg (2005)
12. Baresi, L., Lanzi, P.L., Miraz, M.: TestFul: An evolutionary test approach for Java. In: International Conference on Software Testing, Verification and Validation (ICST), pp. 185–194 (2010)

13. Csallner, C., Smaragdakis, Y., Xie, T.: DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17(8) (2008)
14. d'Amorim, M., Pacheco, C., Xie, T., Marinov, D., Ernst, M.D.: An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In: International Conference on Automated Software Engineering (ASE), pp. 59–68 (2006)
15. Duran, J.W., Ntafos, S.C.: An evaluation of random testing. *IEEE Transactions on Software Engineering (TSE)* 10(4), 438–444 (1984)
16. Frankl, P.G., Weiss, S.N.: An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering (TSE)* 19(8), 774–787 (1993)
17. Galeotti, J., Rosner, N., López Pombo, C., Frias, M.: Analysis of invariants for efficient bounded verification. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 25–36 (2010)
18. Hamlet, D., Taylor, R.: Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering (TSE)* 16(12), 1402–1411 (1990)
19. Inkumsah, K., Xie, T.: Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In: International Conference on Automated Software Engineering (ASE), pp. 297–306 (2008)
20. Myers, G.: *The Art of Software Testing*. Wiley, New York (1979)
21. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: International Conference on Software Engineering (ICSE), pp. 75–84 (2007)
22. Ribeiro, J.C.B., Zenha-Rela, M.A., de Vega, F.F.: Test case evaluation and input domain reduction strategies for the evolutionary testing of object-oriented software. *Information and Software Technology* 51(11), 1534–1548 (2009)
23. Ribeiro, J.C.B., Zenha-Rela, M.A., de Vega, F.F.: Enabling object reuse on genetic programming-based approaches to object-oriented evolutionary testing. In: Esparcia-Alcázar, A.I., Ekárt, A., Silva, S., Dignum, S., Uyar, A.Ş. (eds.) EuroGP 2010. LNCS, vol. 6021, pp. 220–231. Springer, Heidelberg (2010)
24. Schuler, D., Zeller, A.: Javalanche: Efficient mutation testing for Java. In: Symposium on The Foundations of Software Engineering (FSE), pp. 297–298 (2009)
25. Sharma, R., Gligoric, M., Jagannath, V., Marinov, D.: A comparison of constraint-based and sequence-based generation of complex input data structures. In: Workshop on Constraints in Software Testing, Verification and Analysis (CSTVA 2010), pp. 337–342 (2010)
26. Staats, M., Pasareanu, C.: Parallel symbolic execution for structural test generation. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 183–194 (2010)
27. Tonella, P.: Evolutionary testing of classes. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 119–128 (2004)
28. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25(2), 101–132 (2000)
29. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 97–107 (2004)
30. Visser, W., Pasareanu, C.S., Pelànek, R.: Test input generation for Java containers using state matching. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 37–48 (2006)
31. Wappler, S., Wegener, J.: Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In: Genetic and Evolutionary Computation Conference (GECCO), pp. 1925–1932 (2006)

32. Weyuker, E.J., Jeng, B.: Analyzing partition testing strategies. *IEEE Transactions on Software Engineering (TSE)* 17(7), 703–711 (1991)
33. Xie, T., Marinov, D., Notkin, D.: Rostra: A framework for detecting redundant object-oriented unit tests. In: International Conference on Automated Software Engineering (ASE), pp. 196–205 (2004)
34. Xie, T., Marinov, D., Schulte, W., Notkin, D.: Symstra: A framework for generating object-oriented unit tests using symbolic execution. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 365–381. Springer, Heidelberg (2005)

Seamless Testing for Models and Code*

Andreas Holzer¹, Visar Januza j², Stefan Kugele³, Boris Langer⁴,
Christian Schallhart⁵, Michael Tautschnig¹, and Helmut Veith¹

¹ Vienna University of Technology, Austria
`{holzer, tautschnig, veith}@forsyte.at`

² TU Darmstadt, Germany
`januzaj@forsyte.de`

³ TU München, Germany
`kugele@in.tum.de`

⁴ Diehl Aerospace GmbH, Germany
`boris.langer@diehl-aerospace.de`

⁵ Oxford University Computing Laboratory, UK
`christian.schallhart@comlab.ox.ac.uk`

Abstract. This paper describes an approach to model-based testing where a test suite is generated from a model and automatically concretized to drive an implementation. Motivated by an industrial project involving DO-178B compliant avionics software, where the models are UML activity diagrams and the implementation is ANSI C, we developed a seamless testing environment based on our test specification language FQL. We demonstrate how to apply FQL to activity diagrams in such a way that FQL test specifications easily translate from UML to C code. Our approach does not require any additional glue or auxiliary code but is fully automatic except for straightforward source code annotations that link source and model. In this way, we can check for modeled but unimplemented behavior and vice versa, and we can also evaluate the degree of abstraction between model and implementation.

1 Introduction

In most industries, testing is the predominant approach to check the correctness of a system under development. The main challenge in testing is to establish an efficient procedure for the selection of useful test cases. Manual testing, appropriately done, requires both expertise and significant effort, and is therefore often either too imprecise or too expensive. Test automation, on the other hand, needs to incorporate domain knowledge to guide the selection of test cases. We are therefore seeing a long term trend towards model-based testing techniques where engineers provide models from which the test cases are derived.

* Supported by BMWI grant 20H0804B in the frame of LuFo IV-2 project INTECO, by DFG grant FORTAS - Formal Timing Analysis Suite for Real Time Programs (VE 455/1-1), and by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007–2013) / ERC grant agreement no. 246858.

The usefulness of a set of test cases, a *test suite*, is naturally correlated to its impact on the system requirements. Thus, development guidelines such as DO-178B [1] insist that the test suite has to cover all system requirements. Model-based testing therefore needs formalisms which allow us to translate informal textual requirements into models. The modeling language typically involves UML-style automata concepts for control-centric software, or pre- and postcondition descriptions for data-centric computations. While the translation of requirements into models is done by a human, the subsequent steps lend themselves to automation. In particular, the translation of requirements into models enables us to formalize and operationalize the hitherto informal notion of “requirement coverage”: Coverage can be measured relative to model entities, e.g., as coverage of model states. Requirement coverage is thus becoming an algorithmic question.

In a typical model-based testing tool chain, abstract test cases are generated at model level, and then concretized and evaluated on the system under test (SUT). The concretization step – i.e., the translation of abstract test cases to concrete ones – is the most difficult one, as it requires formal models that carry sufficient semantic information for a seamless translation on the one hand, and a suitable testing mechanism for the system under test on the other hand. This mechanism will typically either adapt the SUT to the model level by providing a matching high-level API or by employing test scripts that drive the SUT; it may also combine both methods. In the evaluation step, the achieved source code coverage is observed and failures in the program behavior are checked for. Figure 1 summarizes this basic model-based testing work flow.

Despite its success in both academia and industrial practice, model-based testing has not realized its full potential yet:

- *Requirement coverage on the model is often lacking a precise definition, and only implicitly defined by existing tool chains.* Most available tool chains only support specific hard-coded coverage criteria, such as node or transition coverage. We need a more flexible requirement specification formalism along with tool support to help the test engineer develop the test suite incrementally, to tailor test specifications for relevant goals and to deal with incomplete implementations and/or evolving requirements.
- *Test concretization is typically based on manually crafted test scripts.* Besides being error-prone, inflexible, and expensive, the manually crafted adaption

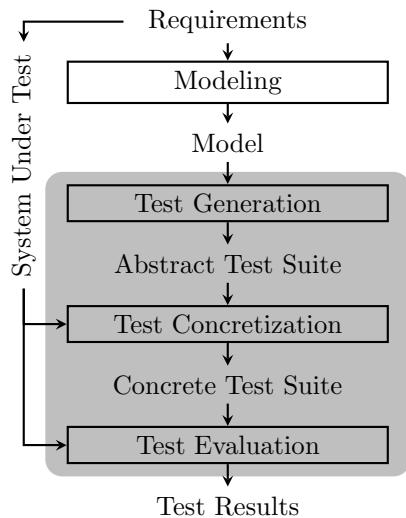


Fig. 1. Model-based testing process

code may introduce hard-to-detect errors and jeopardizes the formal traceability of requirements.

In this paper, we describe a seamless framework for model-based testing which addresses these issues:

- (a) *Versatile Coverage Specifications.* In Section 2, we demonstrate on the example of UML activity diagrams that our test specification language FQL [2] (which was previously used for ANSI C source code) is a versatile, simple, and precise formalism to specify model-level coverage criteria.
- (b) *Automated Test Generation.* In Section 3.1, we show how to use our test input generator FSHELL [3,4] to automatically compute model-level test cases in accordance with the coverage specifications.
- (c) *Automated Test Concretization.* In Section 3.2, we use the model-level test cases as patterns for concrete test cases. Driven by these patterns, FSHELL automatically computes test inputs for the implementation. Thus, we replace hand-written test scripts by a highly automated seamless procedure.
- (d) *Improved Traceability.* Our concretization mechanism is tied to a traceability relation between models and implementation which is based on simple (but necessarily manual) FQL-based annotations. Section 3.3 shows how to compute model/implementation inconsistencies using this relation.
- (e) *Applicability in DO-178B Processes.* We study the applicability of our testing approach to DO-178B compatible system development processes in Section 4.

We developed a plug-in for TOPCASED [5] that implements our methodology for models given as activity diagrams and systems under test written in ANSI C.

2 Seamless Test Specifications in FQL

We first review the concepts of FQL, a coverage specification language originally designed for coverage criteria in imperative programming languages such as ANSI C. For a detailed and complete description of FQL cf. [2]. In Section 2.2 we adapt FQL to support model-based testing and exemplify this step on UML activity diagrams.

2.1 FQL in a Nutshell

In FQL, programs are represented by control flow automata (CFA) [6], which are essentially control flow graphs, bearing the statement labels on their edges instead of their nodes. Figure 2 shows an ANSI C function that returns the maximum value of two given parameters, and below, its CFA. A condition, e.g., $x \geq y$ at Line 4, is modeled by two edges, one for each evaluation of the condition: Edge (4, 5) assumes that $x \geq y$ holds whereas edge (4, 7) assumes $x < y$. The nodes, edges, and paths in a CFA form the potential test targets, e.g., if we want a test suite that covers all statements, we need to reach each CFA node via some test input, while for condition coverage, we need to reach each edge representing the outcome of a condition (edges (4, 5) and (4, 7) in our example).

To specify test targets in FQL, we use *filter functions*. Each filter function calculates a subgraph of a given CFA, e.g., the filter function `ID` computes the identity function. There are also filter functions referring to code structures needed by standard coverage criteria, such as basic block coverage or condition coverage: `@BASICBLOCKENTRY` yields all basic block entries and `@CONDITIONEDGE` yields all evaluations of conditions. In the example above, `ID` yields the CFA itself, `@BASICBLOCKENTRY` yields the subgraph containing all nodes of the CFA and the CFA edges $(1, 4)$, $(5, 9)$, $(7, 9)$, and $(9, 10)$, whereas `@CONDITIONEDGE` yields the subgraph containing the nodes 4, 5, and 7 and the CFA edges $(4, 5)$ and $(4, 7)$. The filter function `@LABEL(L)` refers to the CFA edge that represents the source code annotated with code label `L`, e.g., CFA edge $(7, 9)$ in the example above. We can also refer to the entry and exit edges of the function `max`, i.e., the edges $(1, 4)$ and $(9, 10)$, respectively, by using the expressions `@ENTRY(max)` and `@EXIT(max)`, respectively. Filter functions encapsulate the programming language dependent part of FQL; all further aspects, as described below, are *independent* of the programming language.

Using the operators `NODES`, `EDGES`, and `PATHS`, we select the nodes, edges, or paths in the subgraph identified by filter functions. For example, `NODES(ID)` refers to the CFA nodes 1, 4, 5, 7, 9, and 10, and the expression `EDGES(@LABEL(L))` refers to the singleton set containing the CFA edge $(7, 9)$. The operator `PATHS(F, k)` takes a filter function `F` and a positive integer bound `k` as parameters and yields the set of paths in the subgraph identified by `F` which pass no CFA edge more than `k` times. In the example above, `PATHS(ID, 1)` denotes the two sequences $\langle(1, 4), (4, 5), (5, 9), (9, 10)\rangle$ and $\langle(1, 4), (4, 7), (7, 9), (9, 10)\rangle$.

To build patterns from these node, edge, and path sets, we recombine these sets into patterns with the standard regular expression operators, i.e., ‘.’, ‘+’, and ‘*’, denoting concatenation, alternative, and the Kleene star. In the following we refer to these patterns as *path patterns*. For example, let `Q` denote the expression `EDGES(ID)*.EDGES(@CONDITIONEDGE).EDGES(ID)*`, then, evaluated on the CFA in Figure 2, `Q` specifies the paths that enter a condition edge after finitely many CFA edges, i.e., either edge $(4, 5)$ or $(4, 7)$, and, finally, reach the program exit after finitely many further steps. Since `EDGES` is used in most cases, FQL allows to omit it, i.e., the pattern above can be abbreviated as `ID*.@CONDITIONEDGE.ID*`. Moreover, the sets constructed with `NODES`, `EDGES`, and `PATHS` may be further qualified with predicates, e.g., the path pattern `ID*.{x > 10}.@LABEL(L).ID*` requires that code label `L` is reached at least once when variable `x` is greater than 10.

```

1 int max(int x, int y) {
2     int tmp;
4     if (x >= y)
5         tmp = x;
6     else
7 L:   tmp = y;
9     return tmp;
10 }
```

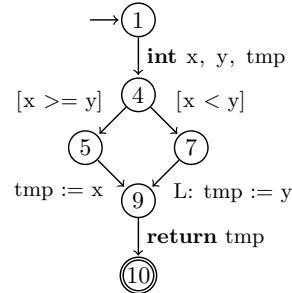


Fig. 2. Sample C function and corresponding CFA

Due to the Kleene star operator in the example expression Q , its language contains infinitely many words, given as finite sequence of predicates, CFA nodes, and CFA edges (here, we consider a path as a sequence of edges). However, when testing, we need a finite set of test targets, and therefore, FQL distinguishes between *path patterns* and *coverage patterns*: Both are regular expressions built from predicates, the operators `NODES`, `EDGES`, `PATHS`, and the concatenation and alternative operators ‘.’ and ‘+’. However, only path patterns are allowed to use the Kleene star ‘*’, while coverage patterns are allowed to *encapsulate* path patterns. Such path patterns are stated in quotes and match an execution fragment that satisfies *one of the words* in the language of the path pattern. For example, in contrast to the infinite language of Q , the language of the coverage pattern “`EDGES(ID)*.EDGES(@CONDITIONEDGE).EDGES(ID)*`” contains only two words: One that requests a program execution that passes after finitely many steps the edge $(4, 5)$ and proceeds with finitely many further steps to the program exit, as well as the analogous word with edge $(4, 7)$ instead of $(4, 5)$.

An FQL query has the general form `cover C passing P`, where C is a coverage pattern and P is a path pattern. It requires a test suite which (i) contains for each word in C at least one matching test case, and (ii), contains only test cases matched by P . For example, to achieve condition coverage with the constraint that each test case reaches code label `L` while $x > 10$ holds, we use the query

```
cover "ID*".@CONDITIONEDGE."ID*" passing ID*.{x > 10}.@LABEL(L).ID*
```

The `passing` clause is optional and defaults to `passing ID*` upon omission.

2.2 FQL for UML Models

The graphical representation of transition-based UML modeling formalisms like UML activity diagrams or UML state machines [7] lend themselves to an interpretation as control flow automata. Since these diagrams use a different semantics for their nodes and edges than FQL for its CFA, we have to define new filter functions. As stated above, filter functions are the interface of FQL to different programming and modeling formalisms, hence, apart from filter functions, the definitions of path and coverage patterns remain unchanged. Using UML activity diagrams, we exemplify the application of FQL to UML models: Figure 3 shows an example diagram \mathcal{M} , where the behaviors of the action nodes and guards are given informally, i.e., as plain text. The diagram describes the printing functionality for elements of a linked list. Guarded by the assertion that the input list is not empty, the head of the list is selected and printed, then its successor is processed, and so forth, until every element of the list has been printed. \mathcal{M} contains all node types currently supported by our TOPCASED plug-in: *Action* (\square), *decision* and *merge* (\diamond), *initial* (\bullet), and *activity final nodes* (\circlearrowleft).

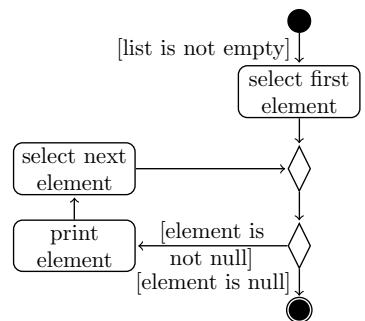


Fig. 3. Model \mathcal{M} : Printing a list

Table 1. Coverage criteria for model-based testing stated as FQL queries

Coverage Criterion	FQL Query
All-states Coverage	"ID*".NODES(ID). "ID*"
All-transitions Coverage	"ID*".ID. "ID*"
All-transition-pairs Coverage	"ID*".ID.ID. "ID*"

We consider an activity diagram as a finite automaton whose transitions can be labeled with guards and whose activity nodes are labeled with operations describing the behavior of the node. Guards and operations can be plain text, as in the example, or be formal with an exact semantics like UML OCL expressions. The operation of a call-behavior node is given as an associated subdiagram. The semantics of the operators NODES, EDGES, and PATHS does not change and, therefore, we can express standard coverage criteria for model-based testing immediately as FQL specifications. Table 1 gives FQL queries for *all-states coverage*, *all-transitions coverage*, and *all-transition-pairs coverage* (cf. [8]). The simplest criterion, all-states coverage, requires every node in the model to be covered by a test case. All-transitions coverage requests a test suite that covers every transition in the model. Finally, all-transition-pairs coverage requires a test case for each pair of consecutive transitions in the model. Besides standard coverage criteria, we are also able to express coverage criteria specific to the model for which we want to generate tests. This enables us, e.g., to prioritize test generation with respect to most critical features of the system being developed. Furthermore, the test designer can specify semantic information present in the model as text only, in the FQL query. For example, the query

```
cover "ID*".ID. "ID*" passing ID*.NODES(@ACTION(print element)).ID*
```

requires transition coverage for the model in Figure 3 where the action *print element* is reached at least once encoding the informally specified constraint that only nonempty lists are possible as inputs.

Transitions in an activity diagram are always labeled with a guard (defaulting to *true*, if blank). For simple conditions, i.e., no Boolean combinations of predicates, we consider the guarded transition as a CFA edge labeled with an assume statement as introduced in Section 2.1. More complex guards, however, result in condition graphs which contain nodes and edges for all involved primitive conditions. For example, an expression PATHS(@GUARDS, 1) denotes all paths in the condition graphs resulting from guards. Note, as UML OCL constraints do not contain loops, the bound 1 is sufficient to refer to all possible paths inside the condition graph resulting from such a guard. So, to specify a test suite that simultaneously achieves path coverage for all guard conditions and covers all nodes in the activity diagram, we state the FQL query `cover "ID*".(NODES(@ACTIVITYNODES) + PATHS(@GUARDS, 1))."ID*"`. Thus, we are able to refer to elements of the graphical representation of an activity diagram as well as to structural elements of the guards within the same formalism, i.e., an FQL query. In Section 3.1, we show how we generate test cases for FQL specifications stated on activity diagrams.

3 Test Process

We present our test process following its steps *test generation* (Section 3.1), *test concretization* (Section 3.2), and *test evaluation* (Section 3.3) as depicted in Figure 1 and use the example diagram \mathcal{M} given in Figure 3 for illustration.

3.1 Test Generation

Via a *generation query* Q_g we specify the coverage we want to achieve at model level. For example, the query `cover PATHS(ID, 1)` requires a test suite where all loops in \mathcal{M} are either skipped or traversed once. We realize the generation of model-level tests by translating \mathcal{M} into a C program \mathcal{M}' and Q_g into an FQL specification Q'_g . Then, we use FSHELL to generate a test suite for \mathcal{M}' that achieves the coverage required by Q'_g .

Listing 1 shows the C code generated from \mathcal{M} . Each diagram node corresponds to a code label and a call to a logging mechanism, e.g., the code labeled with `PL_ENTRY` corresponds to the initial node in Figure 3. When program execution reaches this code, we log the unique identifier `PL_ENTRY_ID` of the initial node. Flows between two nodes are realized via `goto` statements. In case of a decision node, a `switch` structure realizes the branching control flow. We use exactly one C function per activity diagram and realize calls to subdiagrams as function invocations (there are none in this example). The function ‘decision’, which is declared but not defined, controls the flow. On evaluating Q'_g on \mathcal{M}' , FSHELL generates the definition of the function ‘decision’ as a representation of the computed test suite: For the running example, the generated definition is shown in Listing 2. We use the global variable `_fshell2_tc_selector` to choose one of the generated test cases, such that ‘decision’ returns the sequence of decisions necessary to guide the execution through the selected test case. After compiling and linking \mathcal{M}' together with the generated function ‘decision’, the execution of the resulting program produces a log which identifies the model elements passed during execution. Thereby, we obtain a model-level test case $C_{\mathcal{M}}^i$ as a sequence of model elements. Figure 4 depicts the model-level test suite $S_{\mathcal{M}} = \{C_{\mathcal{M}}^1, C_{\mathcal{M}}^2\}$ generated for the query $Q_g = Q'_g = \text{cover } @\text{PATHS}(ID, 1)$. After inspecting $S_{\mathcal{M}}$, the test engineer either releases the suite or adjusts Q_g to enhance the suite until achieving requirements coverage on the model.

A model with an executable semantics can improve the test generation step by encoding the formal semantics of the model elements into the generated C program, such that the generated test cases are more meaningful and spurious test cases can be avoided. We can encode guards with a formal semantics into the generated C code with FSHELL’s support for assumptions: FSHELL supports a C function `_CPROVER_assume` with a semantics analogous to a guard, i.e., FSHELL considers only those program executions which do not violate any assumption.

3.2 Test Concretization

In order to concretize model-level test cases, such as those shown in Figure 4, we need to relate model entities with source code elements. This relation yields

```

1 // FShell generates this function
2 extern int decision();

4 void diagram_M() {
5 // initial
6 PL_ENTRY: log(PL_ENTRY_ID);
7 goto PL_1;
8 // "list is not empty" has no code

10 // select first element
11 PL_1: log(PL_1_ID); goto PL_2;
12 // merge node
13 PL_2: log(PL_2_ID); goto PL_3;
14 // decision node
15 PL_3: log(PL_3_ID);
16 switch (decision()) {
17 // element is not null
18 case 0: goto PL_4;
19 // element is null
20 default: goto PL_EXIT;
21 }
22 // print element
23 PL_4: log(PL_4_ID); goto PL_5;
24 // select next element
25 PL_5: log(PL_5_ID); goto PL_2;
26 // activity final
27 PL_EXIT: log(PL_EXIT_ID);
28 }

30 int main() {
31 diagram_M();
32 return (0);
33 }

```

Listing 1. Generated C program \mathcal{M}'

```

1 extern unsigned _fshell2_tc_selector;
2 int decision(){
3 static unsigned idx = 0;
4 int retval1 [1] = { 8192 };
5 int retval2 [2] = { 0,524288 };
6 switch ( _fshell2_tc_selector ) {
7 case 0: return retval1[idx++];
8 case 1: return retval2[idx++];
9 }
10 }

```

Listing 2. Generated C function decision

Model-level test case $C_{\mathcal{M}}^1$:



Model-level test case $C_{\mathcal{M}}^2$:

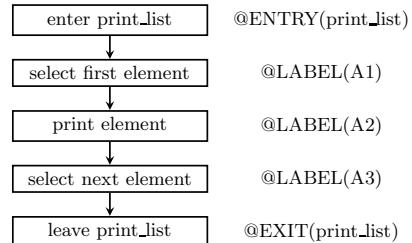


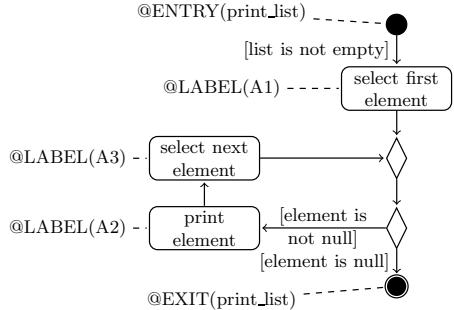
Fig. 4. Model-level test suite $S_{\mathcal{M}} = \{C_{\mathcal{M}}^1, C_{\mathcal{M}}^2\}$ with links to source code (see Section 3.2)

additional traceability information between model and implementation, which we exploit in the test evaluation described in Section 3.3.

Continuing the example, we use the hand-coded program in Listing 3 as implementation of the model shown in Figure 3 and, already annotated, in Figure 5: To establish links between model and source code, we rely on FQL's expressiveness in referring to specific implementation elements, and use annotations consisting of two parts: (i) Labels added to the relevant source code locations, and (ii) matching annotations at activity diagram nodes. As shown in Figure 5, we associate model elements with FQL filter expressions and thereby refer to the SUT via the corresponding labels. For example, we link the action *select first element* with the filter function expression $@LABEL(A1)$ which identifies the assignment labeled with $A1$ in Listing 3. Entry and exit nodes of activity diagrams are annotated with $@ENTRY(print_list)$ and $@EXIT(print_list)$, respectively.

```

1 void print_list (struct list * p_list ) {
2     struct list_element* cur_elem;
3     assert ( p_list != 0 &&
4             p_list ->head != 0);
5 A1: cur_elem = p_list ->head;
6     while (cur_elem != 0) {
7 A2: if (cur_elem->allocated != 0)
8         printf("ALLOCATED\n");
9     else
10        printf("FREE\n");
11 A3: cur_elem = cur_elem->next;
12 }
13 }
```

Listing 3. Realization of `print_list`**Fig. 5.** Printing elements of a linked list with additional FQL annotations

Building upon this mapping, we automatically translate the model-level paths of S_M into sequences of code labels. From each such sequence, a `passing` clause P_c^i for the implementation is computed such that every matching execution of the implementation concretizes the corresponding model-level test case C_M^i . For a precise description of the desired executions, the code label sequences must be augmented with restrictions on the permitted statements *between* points of code prescribed by the model-level path. Consider, e.g., the model-level test case C_M^1 : After *select first element* (i) no other action node must be reached and (ii) function `print_list` must not be left and/or re-entered before reaching *leave print_list*. These properties are best described using FQL's set-theoretic operations *set-complement*, "NOT", and *set-union*, "I". We use FSHELL's C-style macro feature and derive from the model M a suitable definition of a macro `m_nodes` to denote the set of all model-level entities. For our running example `m_nodes` is defined to be `@LABEL(A1) | @LABEL(A2) | @LABEL(A3) | @ENTRY(print_list) | @EXIT(print_list)`. We then use this macro to describe the desired restriction as "NOT(`m_nodes`)*". For example, C_M^1 concretizes to P_c^1 :

$$P_c^1 = \text{passing } @ENTRY(\text{print_list}) . \text{"NOT}(\text{m_nodes})*\text{"} . @LABEL(\text{A1}) \\ . \text{"NOT}(\text{m_nodes})*\text{"} . @EXIT(\text{print_list})$$

To compute the concrete (implementation level) test suite S_I , we use a *concretization query* Q_c , e.g., $Q_c = \text{cover } \text{"ID*".@CONDITIONEDGE."ID*"}.$ We compute for each `passing` clause P_c^i a concrete test suite S_I^i by evaluating Q_c^i , which combines Q_c and P_c^i . Thus, FSHELL produces for each model-level test case C_M^i with S_I^i a test suite which *covers as many test goals of Q_c as possible such that its test cases only follow P_c^i* . For $Q_c = \text{cover } \text{"ID*".@CONDITIONEDGE."ID*"}.$ this results in covering all branches along the path prescribed by P_c^i . In our example, for the model-level test case C_M^1 we obtain:

$$Q_c^1 = \text{cover } \text{"ID*".@CONDITIONEDGE."ID*"} \text{ passing } @ENTRY(\text{print_list}) \\ . \text{"NOT}(\text{m_nodes})*\text{"} . @LABEL(\text{A1}) . \text{"NOT}(\text{m_nodes})*\text{"} . @EXIT(\text{print_list})$$

We pass the source code of the SUT, macro definitions, and Q_c^i to FSHELL to compute S_I^i . Depending on the relationship between model and implementation,

each generated suite $S_{\mathcal{I}}^i$ may contain none, one, or several concrete test cases, i.e., $S_{\mathcal{I}}^i = \{C_{\mathcal{I}}^{i,1}, \dots, C_{\mathcal{I}}^{i,k_i}\}$, where $k_i \geq 0$ denotes the size of $S_{\mathcal{I}}^i$. The final executable test suite $S_{\mathcal{I}}$ is the union $\bigcup_i S_{\mathcal{I}}^i$ of all individual test suites $S_{\mathcal{I}}^i$. For our example and Q_c^1 , FSHELL finds all test goals to be infeasible, i.e., $S_{\mathcal{I}}^1 = \emptyset$. For Q_c^2 , however, FSHELL will return two test inputs:

$$S_{\mathcal{I}}^2 = \{C_{\mathcal{I}}^{2,1}, C_{\mathcal{I}}^{2,2}\} = \{ * \text{p_list} = \{.\text{head} = \{.\text{allocated} = 0, .\text{next} = \text{NULL}\}\}, \\ * \text{p_list} = \{.\text{head} = \{.\text{allocated} = 1, .\text{next} = \text{NULL}\}\} \}$$

All model level test cases should concretize to singleton sets; we study the reasons for the deviations occurring in our running example in the next section.

3.3 Test Evaluation

Finally, we analyze the implementation-level test suite $S_{\mathcal{I}}$ to identify mismatches in the relationship between model and implementation and execute the test cases in $S_{\mathcal{I}}$ to find errors in the implementation. We categorize the potentially occurring deviations into the deficiencies **(D1)** to **(D4)**, as discussed below. Depending on the quality assurance and certification constraints to be obeyed, some of these deficiencies are perfectly acceptable while others require an update of implementation, model, or both.

Concretization Deficiencies. First, we consider the deficiencies which are identifiable from the structure of $S_{\mathcal{I}}$ and its constituent suites $S_{\mathcal{I}}^i = \{C_{\mathcal{I}}^{i,1}, \dots, C_{\mathcal{I}}^{i,k_i}\}$. Please recall the mapping between model and implementation, as established by our annotations: Each implementation construct is ideally labeled with the corresponding node from the model and vice versa. If this is the case, each model-level test case $C_{\mathcal{M}}^i$ yields a test suite $S_{\mathcal{I}}^i$ with exactly one test case $C_{\mathcal{I}}^{i,1}$. Otherwise, we observe one of the following two deficiencies.

- *Implementation Poverty (D1):* There is a test suite $S_{\mathcal{I}}^i$ with $|S_{\mathcal{I}}^i| = 0$.

Poverty occurs when a model-level test case $C_{\mathcal{M}}^i$ does not yield any concrete test case. Then either incomplete implementations are tested, or the model over-approximates the implementation behavior. For our example and the model-level test suite shown in Figure 4, the first query Q_c^1 yields an empty test suite $S_{\mathcal{I}}^1$, i.e., we observe **(D1)**. Our model cannot formally describe control conditions and thus the model-level test case generation does not consider the fact that `print_list` processes only nonempty lists. This precondition is enforced in Listing 3, where we assert `p_list ->head != 0`, such that `cur_elem` is initialized at label A1 with a non-zero value. Therefore, the condition of the **while** loop cannot evaluate to false and the loop body is entered at least once.

- *Implementation Liberty (D2):* There is a test suite $S_{\mathcal{I}}^i$ of size $|S_{\mathcal{I}}^i| > 1$.

Liberty occurs, if the activities visited by $C_{\mathcal{M}}^i$ are uncoverable with a single concrete test case. This happens whenever the model is more abstract than the implementation, where the precise meaning of “more abstract” depends on the coverage criteria employed; mostly, it means that some model activities necessitate non-trivial control flow in the implementation. Depending on the quality

constraints, this can be perfectly acceptable. However, in critical systems, such as DO-178B compliant software, undocumented code is not allowed (cf. [1], §6.3.4) and liberty indicates a violation thereof.

In our example, as discussed in Section 3.2, $S_{\mathcal{I}}^2$ contains two concrete test cases, revealing liberty in the action *print element*. Both test cases contain a list with a single element where its field *allocated* is either initialized with 0 or 1. The reason for this implementation liberty is the unmodeled **if–then–else** construct in the **while** loop of Listing 3. We choose to correct this situation by replacing the action *print element* with a call behavior action that is associated with the diagram shown in Figure 6. We also label both, the source code and the respective model entities, to relate them with each other (we do not show the new labels in the listing).

Evaluation Deficiencies. The remaining two deficiencies are discovered by analyzing the concrete test suite $S_{\mathcal{I}}$ —once by checking whether it achieves coverage on the implementation, and once by running it. FSHELL provides support for automatically constructing a test harness from $S_{\mathcal{I}}$. This driver for the SUT enables execution and proper evaluation.

The check for coverage is controlled by the *evaluation query* Q_e : This query is determined by the applicable certification standard, which demands to satisfy a certain structural coverage at implementation level with a test suite achieving requirements coverage at model level, i.e., Q_e amounts to an adequacy criterion for the generated test suite. For example, DO-178B Design Assurance Level C requires statement coverage, i.e., Q_e would be `cover "ID*".NODES(ID)."ID*"`.

- *Implementation Anarchy (D3)*: The test suite $S_{\mathcal{I}}$ does not satisfy the *evaluation query* Q_e on the implementation.

Anarchy relative to Q_e occurs when the implementation is not fully modeled, requiring—depending on the applicable certification standard—either corrections in the model and/or implementation, or a rationale explaining the omission, e.g., if third-party code is involved. Given the expressiveness of FQL, such omissions can be formally documented by adjusting Q_e to require only coverage of code fragments which are relevant to this development process.

- *Implementation Error (D4)*: The implementation exhibits erroneous behavior on executing the test suite $S_{\mathcal{I}}$.

The last deficiency, implementation error, occurs upon assertion violations and unexpected program outputs, and requires in all likelihood a correction to bring implementation and model into mutual correspondence. While checking assertions does not require any further provisions, error detection necessities test

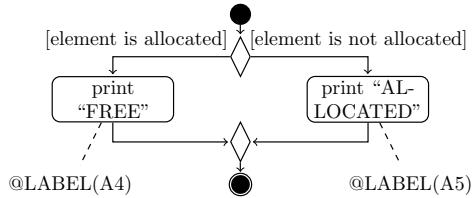


Fig. 6. Activity diagram of **if** structure

oracles for monitoring the program output. If the model lacks an executable semantics, as in our example, oracles could only be obtained from additional annotations given by the user (editing the automatically generated SUT driver is possible, but undesirable). For models with an executable semantics, the guards translate to assertions, which serve as test oracles in the generated SUT driver.

Seamless Traceability. Note that if implementation poverty, liberty, and anarchy (**D1-3**) do not occur in a testing process, then model and source code implement essentially the same control flow. Moreover, the annotations used in model and implementation precisely document the relationship between them and enable the mutual traceability of requirements and program features.

4 Prototype Evaluation

In our joint project INTECO, we used our TOPCASED plug-in to apply our approach to a memory manager which is part of a helicopter pilot assistance and mission planning system. This memory manager is implemented in 526 lines of ANSI C code¹ and should avoid memory fragmentation caused by dynamic allocations. To do so, it provides an API that enables the programmer to gather individual chunks of memory in a statically acquired memory area. We studied the feasibility of our approach using UML activity diagrams describing the behavior of API functions. The derived requirements for the memory manager yielded 21 activity diagrams, 17 of them are implemented as C functions, two are part of these functions, and the remaining two are macros. Amongst these diagrams, there were at most 13 action nodes, two decision nodes, and one loop in an activity. Albeit small in size, our prototypical case study is based on an industrial avionics software component, demonstrating the capability of our tool chain to deal with real-world C code.

The initial concretization revealed several errors in design and coding: The order of the action nodes of one activity diagram was not correctly reflected in the C code of the implementation. Thus, both, implementation anarchy (**D3**) and poverty (**D1**) occurred: Some code remained unreachable in the paths prescribed by the abstract test cases, resulting in (**D3**), and one further test case was impossible to concretize, leading to (**D1**). We did not observe implementation liberty, which was an important aspect as DO-178B compatible development requires full traceability between derived low level requirements (which were here modeled using activity diagrams) and implementation. During execution, no further implementation errors (**D4**) were found. Unfortunately, we cannot publish the model and source code studied here, as it is covered by a non-disclosure agreement. But as soon as possible, we will publish our TOPCASED plugin under an open source license on the web².

¹ Source lines of code (SLOC), measured with David A. Wheeler's SLOCCount tool.

² <http://code.forsyte.de/fshell>

5 Related Work

The basic principles behind model-based testing were described by Chow in 1978 [9], the term model-based testing was coined and further refined by Datal et al. [10]. Their work includes automated test input generation and focuses on boundary value testing.

Most existing formalisms for test specifications focus on the description of test data, e.g., TTCN-3 [11] and the UML 2.0 Testing Profile [12], but none of them allows to describe structural coverage criteria. Friske et al. [13] have presented coverage specifications using OCL constraints. Although OCL provides the necessary operations to speak about UML models, it has not been intended for coverage specifications, and henceforth, complex coverage specifications might be hard to express and read. At the time of publication, no tool support for the framework in [13] has been reported. Hessel et al. [14] present a specification language for model-level coverage criteria that uses parameterized observer automata. Test suites for coverage criteria specified in this language can be generated with UPPAAL COVER [15].

The approach of [16] is similar to our work but targeted at Java programs: Their focus lies on automatic test execution, observation of traces, and test input generation. The latter is, however, performed using program specific generators. In [17], the generation of test inputs for Simulink models is realized via a translation of models to C code. This code is subsequently processed by a tool which is—like FSHELL—built upon CBMC [18]. The AGEDIS project [19] also aims at automating model-based testing. Their *test execution directives* are the necessary adapters to translate model-level tests to executable code. In AGEDIS it is assumed that these directives are part of the input provided by the user. The UniTesK tool chain [20] calls adapters *mediators*, which have to be created (semi-)manually with some wizards. Compared to the T-VEC® tools [21], we focus on UML activity diagrams instead of Simulink models and we support automated test generation for informal models.

One of the most advanced testing tool chains is Spec Explorer [22]. It combines model-based testing with various techniques for automated test case generation. Spec Explorer works on Spec# models and .Net code and uses AsmL [23] as formal foundation. Spec Explorer analyzes a simulation relation between model and implementation and uses a mapping as coarse as functions (which is trivial in our case, because we have a single activity diagram per C function). Spec Explorer includes techniques for test case selection—however, they are not as fine grained as FQL. Building upon Spec Explorer, Kicillof et al. [24] describe an approach that combines model-level black-box testing with parametrized white-box unit testing. They generate unit tests for an extended version of activity diagrams and concretize these tests via white-box test case generation. Their work aims at generating high implementation coverage, while we focus on a DO-178B compatible processes, i.e., we only measure the achieved implementation coverage and check for possible deficiencies.

Black-box approaches, such as input/output conformance (ioco) testing as performed in the TorX framework [25], require a different kind of mapping, which

focuses on interface descriptions. But even in such cases, FSHELL is applicable, albeit we could use a fraction only of its power.

There exist several approaches that cover specifically the test input generation part for UML models: In the tradition of automata-theoretic methods, the most common [26] approaches employ UML state machines [27,28] and interaction diagrams [29], respectively. Test case generation based on activity diagrams for Java programs was introduced by Chen et al. in [30,31,32]: They propose in [30] a method to generate random test cases, and introduce in [31] a coverage-directed approach using the model checker NuSMV [33]. While their first, random-based approach, is unlikely to achieve good coverage, their second approach suffers from the state space explosion problem and appears to be unscalable. Kundu and Samanta [34] present an extension of [31,32] which is aimed at concurrent Java applications and uses much more abstract models leading to test cases which are apparently not executable without additional processing.

6 Conclusion

Many certification standards are demanding tests generated from requirement-derived models, and ask for seamless traceability of low-level requirements. In this paper, we provide a solution to both challenges: First, exploiting the expressiveness and adaptability of FQL, we specify model-level test suites and generate them with FSHELL. Second, relying on annotations of model and source, we concretize the model-level test suites to the implementation. As this step does not involve adaption code but only annotations, it enables us to assess the relation between model and source in a precise manner. Although our example and case study consider low level models, our approach is not limited to that: High-level models can refer to function calls instead of code labels, or both of them.

Our prototype demonstrates that our approach is applicable to industrial projects and does indeed find deficiencies in these examples. We are currently working on a larger case study with industrial collaborators. Future research goals include automatic generation of source code stubs and test oracles.

Acknowledgments

We want to thank the anonymous reviewers for their useful remarks.

References

1. RTCA DO-178B. Software considerations in airborne systems and equipment certification (1992)
2. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: How did you specify your test suite. In: ASE, pp. 407–416 (2010)
3. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FSHELL: systematic test case generation for dynamic analysis and measurement. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 209–213. Springer, Heidelberg (2008)

4. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 151–166. Springer, Heidelberg (2009)
5. Farail, P., Gauillet, P., Canals, A., Le Camus, C., Sciamma, D., Michel, P., Crégut, X., Pantel, M.: The TOPCASED project: a toolkit in open source for critical aeronautic systems design. In: ERTS, pp. 54–59 (2006)
6. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70 (2002)
7. OMG. UML 2.0 Superstructure Specification. Technical Report ptc/04-10-02, Object Management Group (2004)
8. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco (2006)
9. Chow, T.: Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering SE-4(3), 178–187 (1978)
10. Dalal, S.R., Jain, A., Karunanihi, N., Leaton, J.M., Lott, C.M., Patton, G.C., Horowitz, B.M.: Model-based testing in practice. In: ICSE, pp. 285–294 (1999)
11. Din, G.: TTCN-3. In: Model-Based Testing of Reactive Systems, pp. 465–496 (2004)
12. Schieferdecker, I., Dai, Z.R., Grabowski, J., Rennoch, A.: The UML 2.0 testing profile and its relation to TTCN-3. In: Hogrefe, D., Wiles, A. (eds.) TestCom 2003. LNCS, vol. 2644, pp. 79–94. Springer, Heidelberg (2003)
13. Friske, M., Schlingloff, B.-H., Weißleder, S.: Composition of model-based test coverage criteria. In: MBEES, pp. 87–94 (2008)
14. Blom, J., Hessel, A., Jonsson, B., Pettersson, P.: Specifying and generating test cases using observer automata. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 125–139. Springer, Heidelberg (2005)
15. Hessel, A., Larsen, K.G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using UPPAAL. In: FORTEST, pp. 77–117 (2008)
16. Artho, C., Drusinsky, D., Goldberg, A., Havelund, K., Lowry, M.R., Pasareanu, C.S., Rosu, G., Visser, W.: Experiments with test case generation and runtime analysis. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 87–107. Springer, Heidelberg (2003)
17. Brillout, A., He, N., Mazzucchi, M., Kroening, D., Purandare, M., Rüümmer, P., Weissenbacher, G.: Mutation-based test case generation for simulink models. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 208–227. Springer, Heidelberg (2010)
18. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
19. Hartman, A., Nagin, K.: Model driven testing - AGEDIS architecture interfaces and tools. In: First European Conference on Model Driven Software Engineering, pp. 1–11 (2003)
20. Kuliamin, V.V., Petrenko, E.K., Kossatchev, E.S., Bourdonov, I.B.: Unitesk: Model based testing in industrial practice. In: First European Conference on Model Driven Software Engineering, pp. 55–63 (2003)
21. T-VEC (January 2011), <http://www.t-vec.com/>
22. Veines, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-based testing of object-oriented reactive systems with spec explorer. In: FORTEST, pp. 39–76 (2008)

23. Barnett, M., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Towards a tool environment for model-based testing with ASML. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 252–266. Springer, Heidelberg (2004)
24. Kicillof, N., Grieskamp, W., Tillmann, N., Braberman, V.A.: Achieving both model and code coverage with automated gray-box testing. In: A-MOST, pp. 1–11 (2007)
25. Tretmans, J., Brinksma, E.: TorX: Automated model-based tesing. In: First European Conference on Model-Driven Software Engineering, pp. 31–43 (2003)
26. Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H.: A survey on model-based testing approaches: a systematic review. In: WEASEL Tech., pp. 31–36 (2007)
27. Weißleder, S., Schlingloff, B.H.: Deriving input partitions from UML models for automatic test generation. In: Giese, H. (ed.) MODELS 2008. LNCS, vol. 5002, pp. 151–163. Springer, Heidelberg (2008)
28. Chevalley, P., Thévenod-Fosse, P.: Automated generation of statistical test cases from UML state diagrams. In: COMPSAC, pp. 205–214 (2001)
29. Nayak, A., Samanta, D.: Model-based test cases synthesis using UML interaction diagrams. SIGSOFT Softw. Eng. Notes 34(2), 1–10 (2009)
30. Chen, M., Qiu, X., Li, X.: Automatic test case generation for UML activity diagrams. In: AST, pp. 2–8 (2006)
31. Chen, M., Mishra, P., Kalita, D.: Coverage-driven automatic test generation for UML activity diagrams. In: GLSVLSI, pp. 139–142 (2008)
32. Chen, M., Qiu, X., Xu, W., Wang, L., Zhao, J., Li, X.: UML activity diagram-based automatic test case generation for java programs. The Computer Journal 52(5), 545–556 (2009)
33. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new Symbolic Model Verifier. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999)
34. Kundu, D., Samanta, D.: A novel approach to generate test cases from UML activity diagrams. Journal of Object Technology 8(3), 65–83 (2009)

Retrofitting Unit Tests for Parameterized Unit Testing

Suresh Thummalapenta¹, Madhuri R. Marri¹, Tao Xie¹,
Nikolai Tillmann², and Jonathan de Halleux²

¹ Department of Computer Science, North Carolina State University, Raleigh, USA
`{sthumma, mrmarri, txie}@ncsu.edu`

² Microsoft Research, One Microsoft Way, Redmond, USA
`{nikolait, jhalleux}@microsoft.com`

Abstract. Recent advances in software testing introduced *parameterized unit tests* (PUT), which accept parameters, unlike conventional unit tests (CUT), which do not accept parameters. PUTs are more beneficial than CUTs with regards to fault-detection capability, since PUTs help describe the behaviors of methods under test for all test arguments. In general, existing applications often include manually written CUTs. With the existence of these CUTs, natural questions that arise are whether these CUTs can be retrofitted as PUTs to leverage the benefits of PUTs, and what are the cost and benefits involved in retrofitting CUTs as PUTs. To address these questions, in this paper, we conduct an empirical study to investigate whether existing CUTs can be retrofitted as PUTs with feasible effort and achieve the benefits of PUTs in terms of additional fault-detection capability and code coverage. We also propose a methodology, called *test generalization*, that helps in systematically retrofitting existing CUTs as PUTs. Our results on three real-world open-source applications (≈ 4.6 KLOC) show that the retrofitted PUTs detect 19 new defects that are not detected by existing CUTs, and also increase branch coverage by 4% on average (with maximum increase of 52% for one class under test and 10% for one application under analysis) with feasible effort.¹

1 Introduction

Unit tests are widely adopted in software industry for ensuring high quality of production code. Unit testing helps detect defects at an early stage, reducing the effort required in fixing those defects. Recent advances in unit testing introduced parameterized unit tests (PUT) [23], which accept parameters, unlike conventional unit tests (CUT), which do not accept parameters. Existing state-of-the-art test-generation approaches such as Dynamic Symbolic Execution (DSE) [15, 10, 20, 22] can be used in combination with PUTs to automatically generate CUTs by instantiating the parameters. In particular, DSE systematically explores the code under test exercised by a PUT and generates CUTs that achieve high structural coverage such as branch coverage of the code under test. Section 2 presents more details on how CUTs can be generated from PUTs via DSE.

In general, PUTs are more beneficial than CUTs. The primary reason is that PUTs help describe the behaviors of methods under test for all test arguments. With PUTs, test data can be automatically generated using DSE-based approaches, thereby helping

¹ The first and second authors have made equal contributions.

address the following two issues with CUTs. First, developers may not be able to write test data (in CUTs) that exercise all important behaviors of methods under test, thereby resulting in unit tests with low fault-detection capability. Second, developers may write different test data that exercise the same behavior of methods under test, thereby resulting in redundant unit tests [24]. These redundant unit tests increase only the testing time and do not increase the fault-detection capability. Consider the three CUTs shown in Figure 1 for testing the Push method of an integer stack class `IntStack`. These three CUTs exercise the Push method with different test data in different test scenarios. For example, CUT1 and CUT2 exercise Push with different argument values, when the stack is empty, while CUT3 exercises Push, when the stack is not empty. Consider that there is a defect (`Push`) that can be detected by passing a negative value as an argument to `Push`. These three tests cannot detect the preceding defect, since these tests do not pass a negative integer value as an argument. Furthermore, CUT2 is a redundant unit test with respect to CUT1, since `IntStack` has the same behavior for all non-negative integers passed as arguments to `Push`. Since test data is automatically generated by DSE-based approaches that tend to exercise all feasible paths in the methods under test, the *fault-detection capability* of PUTs is often higher than that of CUTs. Furthermore, a single PUT can represent multiple CUTs, thereby reducing the *size of test code* and improve the maintainability of the test code. For example, the PUT shown in Figure 2 tests the same or more behaviors of the method under test as the three CUTs shown in Figure 1.

In general, existing applications often include manually written CUTs [9]. With the existence of these CUTs, natural questions that arise are whether these CUTs can be retrofitted as PUTs to leverage the benefits of PUTs, and what are the cost and benefits involved in retrofitting CUTs as PUTs. Here, cost includes the effort required in retrofitting CUTs as PUTs, and benefits include the additional fault-detection capability and code coverage achieved via retrofitting. However, to the best of our knowledge, there exists no empirical study that shows cost and benefits involved in retrofitting existing CUTs as PUTs. To address this issue, in this paper, we conduct an empirical study to investigate whether existing CUTs can be retrofitted as PUTs with feasible effort and such retrofitting achieves benefits in terms of fault-detection capability and code coverage. We also propose a methodology, called *test generalization*, that includes a systematic procedure for manually retrofitting CUTs as PUTs.

```

01:public void CUT1() {
02:    int elem = 1;
03:    IntStack stk = new IntStack();
04:    stk.Push(elem);
05:    Assert.AreEqual(1, stk.Count());
06:public void CUT2() {
07:    int elem = 30;
08:    IntStack stk = new IntStack();
09:    stk.Push(elem);
10:    Assert.AreEqual(1, stk.Count());
11:public void CUT3() {
12:    int elem1 = 1, elem2 = 30;
13:    IntStack stk = new IntStack();
14:    stk.Push(elem1);
15:    stk.Push(elem2);
16:    Assert.AreEqual(2, stk.Count());
}

```

Fig. 1. Three CUTs test an integer stack that does not accept negative integers

```

01:public void PUT(int[] elem) {
02:    IntStack stk = new IntStack();
03:    foreach (int i in elem) {
04:        stk.Push(i);
05:    }
06:    Assert.AreEqual(elem.Length,
07:                    stk.Count());
}

```

Fig. 2. A single PUT replacing the three CUTs

In particular, our empirical study helps address the following two fundamental questions. First, is it cost-effective to retrofit existing CUTs as PUTs (using test generalization) with regards to the benefits of test generalization? Second, can developers other than the original developers who wrote the code under test (who do not have sufficient knowledge of the code under test) use our methodology to retrofit CUTs as PUTs? Such other developers could be those who take over legacy applications or those who try to augment the test suites for the code not written by them. The primary reason for investigating the second question is that, in general, developers who wrote code under test may not face challenges in writing test oracles (in PUTs) that need to describe the expected behavior for all test arguments; however, these other developers often do not have sufficient knowledge of code under test and may face challenges in writing test oracles in PUTs. Therefore, in this paper, we study whether our test-generalization methodology could help these other developers in addressing the challenge of test-oracle generalization (generalizing test oracles in existing CUTs). In particular, to address this issue, the first and second authors (of this paper) who do not have sufficient knowledge of our applications under analysis follow our methodology to retrofit existing CUTs as PUTs. Our results show that test generalization helps these other developers in achieving additional benefits in terms of fault-detection capability and code coverage with feasible effort.

In summary, this paper makes the following major contributions:

- The first empirical study that investigates cost and benefits involved in retrofitting existing CUTs as PUTs for leveraging the benefits of PUTs.
- A methodology, called *test generalization*, that helps developers to write PUTs with feasible effort by leveraging existing CUTs.
- Our empirical results on three real-world applications (≈ 4.6 KLOC) show that test generalization helps detect 19 new defects that are not detected by existing CUTs, showing the benefits in terms of fault-detection capability with feasible effort. A few of these defects are complex to be detected using manually written CUTs. Furthermore, test generalization increases branch coverage by 4% on average (with a maximum increase of 52% for one class under test and 10% for one application under analysis).

2 Background

We use Pex [4] as an example state-of-the-art DSE-based test generation tool for generating CUTs using PUTs. Pex is a white-box test generation tool for .NET programs. Pex accepts PUTs and symbolically executes the PUTs and the code under test to generate a set of CUTs that can achieve high code coverage of the code under test. Since these generated CUTs are targeted for some common testing frameworks such as NUnit [7], it is possible to debug and analyze failing CUTs. Initially, Pex explores the code under test with random or default values and collects constraints along the execution path. Pex next systematically negates parts of the collected constraints and uses a constraint solver to generate concrete values that guide program execution through alternate paths. Pex has been applied on industrial code bases and detected serious new defects in a software component, which had already been extensively tested previously [22].

3 Test Generalization Methodology

We next present our test generalization methodology that assists developers in achieving test generalization. Although we explain our methodology using Pex, our methodology is independent of Pex and can be used with other DSE-based test generation tools [20]. Our methodology is based on the following two requirements.

- **R1:** the PUT generalized from a passing CUT should not result in false-positive failing CUTs being generated from the PUT.
- **R2:** the PUT generalized from a CUT should help achieve the same or higher structural coverage than the CUT and should help detect the same or more defects than the CUT.

We next describe more details on these two requirements. R1 ensures that test generalization does not introduce false positives. In particular, a CUT generated from a PUT can fail for two reasons: a defect in the method under test (MT) or a defect in the PUT. Failing CUTs for the second reason are considered as false positives. These failing CUTs are generated when generalized PUTs do not satisfy either necessary preconditions of the MT or assumptions on the input domain of the parameters required for passing the test oracle in the PUTs. On the other hand, R2 ensures that test generalization does not introduce false negatives. The rationale is that PUTs provide a generic representation of CUTs, and should be able to guide a DSE-based approach in generating CUTs that exercise the same or more paths in the MT than CUTs, and thereby should have the same or higher fault-detection capability.

We next provide an overview of how a developer generalizes existing CUTs to PUTs by using our methodology to satisfy the preceding requirements and then explain each step in detail using illustrative examples from the NUnit framework [7].

3.1 Overview

Our test generalization algorithm includes five major steps: (S1) *Parameterize*, (S2) *Generalize Test Oracle*, (S3) *Add Assumption*, (S4) *Add Factory Method*, and (S5) *Add*

Algorithm 1 Test Generalization

```

Require: CUTs for an MT M
Ensure: PUTs
1: Set PUTs =  $\emptyset$ , gAllCUTs =  $\emptyset$ 
2: for all  $c \in$  CUTs do
3:   if gAllCUTs.Contains( $c$ ) then
4:     Continue
5:   end if
6:   Set  $p = \emptyset$ , gCUTs =  $\emptyset$ , break = false
7:    $p = \text{Parameterize}(c)$ 
8:    $p = \text{GeneralizeTestOracle}(c, p)$ 
9:   gCUTs =  $\text{GenerateCUTs}(p)$ 
10:  repeat
11:    while !Execute(gCUTs) do
12:      if LegalValueIssue(gCUTs) then
13:         $p = \text{AddAssumption}(p)$ 
14:      else
15:        ReportDefect()
16:        Continue
17:      end if
18:    end while
19:    if ! $\text{Cov}(M, gCUTs) \supseteq \text{Cov}(M, c)$  then
20:      if NPTYPEParam( $p$ ) then
21:         $p = \text{AddFactoryMethod}(p)$ 
22:      end if
23:      if EnviInteractionIssue( $M$ ) then
24:         $p = \text{AddMockObj}(p)$ 
25:      end if
26:      else
27:        break = true
28:      end if
29:    until break
30:    PUTs.Add( $p$ ), gAllCUTs.Add(gCUTs)
31:  end for
32: return PUTs

```

Mock Object. In our methodology, Steps S1 and S2 are mandatory, whereas Steps S3, S4, and S5 are optional and are used when R1 or R2 is not satisfied. Indeed, recent work [21, 16] (as discussed in subsequent sections) could help further alleviate effort required in Steps S3, S4, and S5. We next explain our methodology in detail.

For an MT, the developer uses our algorithm to generalize the set of CUTs of that MT, one CUT at a time. First, the developer identifies concrete values and local variables in the CUT and promotes them as parameters for a PUT (Line 7). Second, the developer generalizes the assertions in the CUT to generalized test oracles in the PUT (Line 8). After generalizing test oracles, the developer applies Pex to generate CUTs (referred to as *gCUTS*) from PUTs (Line 9). When any of the generated CUTs fails (Line 11), the developer checks whether the reason for the failing CUT(s) is due to illegal values generated by Pex for the parameters (Line 12), i.e., whether the failing CUTs are false-positive CUTs. To avoid these false-positive CUTs and thereby to satisfy R1, the developer adds assumptions on the parameters to guide Pex to generate legal input values (Line 13). The developer then applies Pex again and continues this process of adding assumptions till either no generated CUTs fail or the generated CUTs fail due to defects in the MT.

After satisfying R1, the developer checks whether R2 is also satisfied, i.e., the structural coverage achieved by generated CUTs is at least as much as the coverage achieved by the existing CUTs. If R2 is satisfied, then the developer proceeds to the next CUT. On the other hand, if R2 is not satisfied, then there could be two issues: (1) Pex was not able to create desired object states for a non-primitive parameter [21], and (2) the MT includes interactions with external environments [16]. Although DSE-based test-generation tools such as Pex are effective in generating CUTs from PUTs whose parameters are of primitive types, Pex or any other DSE-based tool faces challenges in cases such as generating desirable objects for non-primitive parameters. To address these two issues, the developer writes factory methods (Line 21) and mock objects [16] (Line 24), respectively, to assist Pex. More details on these two steps are described in subsequent sections.

The developer repeats the last three steps till the requirements R1 and R2 are met, as shown in Loop 10-29. Often, multiple CUTs can be generalized to a single PUT. Therefore, to avoid generalizing an existing CUT that is already generated by a previously

```
//MSS=MemorySettingsStorage
00:public class SettingsGroup{
01: MSS storage; ...
02: public SettingsGroup(MSS storage){
03:     this.storage = storage; }
04:     public void SaveSetting(string sn, object sv) {
05:         object ov = storage.GetSetting( sn );
06:         //Avoid change if there is no real change
07:         if(ov != null ) {
08:             if(ov is string && sv is string &&
09:                 (string)ov===(string)sv || |
10:                 ov is int&&sv is int&&(int)ov===(int)sv ||
11:                 ov is bool&&sv is bool&&(bool)ov===(bool)sv ||
12:                 ov is Enum&&sv is Enum&&ov.Equals(sv))
13:                 return;
14:             }
15:             storage.SaveSetting(sn, sv);
16:             if (Changed != null)
17:                 Changed(this, new SettingsEventArgs(sn));
18:     }}
```

Fig. 3. The *SettingsGroup* class of NUnit with the *SaveSetting* method under test

```
00://tg is of type SettingsGroup
01:[Test]
02:public void TestSettingsGroup() {
03:     tg.SaveSetting("X",5);
04:     tg.SaveSetting("NAME","Tom");
05:     Assert.AreEqual(5,tg.GetSetting("X"));
06:     Assert.AreEqual("Tom",tg.GetSetting("NAME"));
07: }
```

Fig. 4. A CUT to test the *SaveSetting* method

generalized PUT, the developer checks whether the existing CUT to be generalized belongs to already generated CUTs (referred to as *gAllCUTs*) (Lines 3 – 5). If so, the developer ignores the existing CUT; otherwise, the developer generalizes the existing CUT. We next illustrate each step of our methodology using an MT and a CUT from the NUnit framework shown in Figures 3 and 4, respectively.

3.2 Example

MT and CUTs. Figure 3 shows an MT `SaveSetting` from the `SettingsGroup` class of the NUnit framework. The `SaveSetting` method accepts a setting name `sn` and a setting value `sv`, and stores the setting in a storage (represented by the member variable `storage`). The setting value can be of type `int`, `bool`, `string`, or `enum`. Before storing the value, `SaveSetting` checks whether the same value already exists for that setting in the storage. If the same value already exists for that setting, `SaveSetting` returns without making any changes to the storage.

Figure 4 shows a CUT for testing the `SaveSetting` method. The CUT saves two setting values (of types `int` and `string`) and verifies whether the values are set properly using the `GetSetting` method. The CUT verifies the expected behavior of the `SaveSetting` method for the setting values of only types `int` and `string`. This CUT is the only test for verifying `SaveSetting` and includes two major issues. First, the CUT does not verify the behavior for the types `bool` and `enum`. Second, the CUT does not cover the `true` branch in Statement 8 of Figure 3. The reason is that the CUT does not invoke the `SaveSetting` method more than once with the same setting name. This CUT achieves 10% branch coverage² of the `SaveSetting` method. We next explain how the developer generalizes the CUT to a PUT and addresses these two major issues via our test generalization.

S1 - Parameterize. For the CUT shown in Figure 4, the developer promotes the `string` “Tom” and the `int` 5 as a single parameter of type `object` for the PUT. The advantage of replacing concrete values with symbolic values (in the form of parameters) is that Pex generates concrete values based on the constraints encountered in different paths in the MT. Since `SaveSetting` accepts the parameter of type `object` (shown in Figure 5), Pex automatically identifies the possible types for the `object` type such as `int` or `bool` from the MT and generates concrete values for those types, thereby satisfying R2. In addition to promoting concrete values as parameters of PUTs, the developer promotes other local variables such as the receiver object (`tg`) of `SaveSetting` as parameters. Promoting such receiver objects as parameters can help generate different object states (for those receiver objects) that can help cover additional paths in the MT. Figure 5 shows the PUT generalized from the CUT shown in Figure 4.

S2 - Generalize Test Oracle. The developer next generalizes test oracles in the CUT. In the CUT, a setting is stored in the storage using `SaveSetting` and is verified using `GetSetting`. By analyzing the CUT, the developer generalizes the test oracle of the

² We use NCover (<http://www.ncover.com/>) to measure branch coverage. NCover uses .NET byte code instead of source code for measuring branch coverage.

CUT by replacing the constant value with the relevant parameter of the PUT. The test oracle for the PUT is shown in Line 4 of Figure 5.

In practice, generalizing a test oracle is a complex task, since determining the expected output values for all the generated inputs is not trivial. Therefore, to assist developers in generalizing test oracles, we proposed 15 PUT patterns, which developers can use to analyze the existing CUTs and generalize test oracles. More details of the patterns are available in Pex documentation [6].

S3 - Add Assumption. A challenge faced during test generalization is that Pex or any DSE-based approach requires guidance in generating legal values for the parameters of PUTs. These legal values are the values that satisfy preconditions of the MT and help set up test scenarios to pass test assertions (i.e., test oracles). These assumptions help avoid generating false-positive CUTs, thereby satisfying R1. For example, without any assumptions, Pex by default generates illegal `null` values for non-primitive parameters such as `st` of the PUT shown in Figure 5. To guide Pex in generating legal values, the developer adds sufficient assumptions to the PUT. In the PUT, the developer annotates each parameter with the tag `PexAssumeUnderTest`³, which describes that the parameter should not be `null` and the type of generated objects should be the same as the parameter type. The developer adds further assumptions to PUTs based on the behavior exercised by the CUT and the feedback received from Pex. Recently, there is a growing interest towards a new methodology, called *Code Contracts* [5], where developers can explicitly describe assumptions of the code under test. We expect that effort required for Step S3 can be further reduced when the code under test includes contracts.

S4 - Add Factory Method. In general, Pex (or any other existing DSE-based approaches) faces challenges in generating CUTs from PUTs that include parameters of non-primitive types, since these parameters require method-call sequences (that create and mutate objects of non-primitive types) to generate desirable object states [21]. These desirable object states are the states that are required to exercise new paths or branches in the MT, thereby to satisfy R2. For example, a desirable object state to cover the `true` branch of Statement 8 in Figure 3 is that the `storage` object should already include a value for the setting name `sn`. Recent techniques in object-oriented testing [21, 13] could

```
//PAUT: PexAssumeUnderTest
00: [PexMethod]
01:public void TestSave([PAUT] SettingsGroup st,
02:                      [PAUT] string sn, [PAUT] object sv) {
03:    st.SaveSetting(sn, sv);
04:    PexAssert.AreEqual(sv, st.GetSetting(sn));
}
```

Fig. 5. A PUT for the CUT shown in Figure 4

```
//MSS: MemorySettingsStorage (class)
//PAUT: PexAssumeUnderTest (Pex attribute)
00: [PexFactoryMethod(typeof(MSS))]
01:public static MSS Create([PAUT]string[]
02:                           sn, [PAUT]object[] sv) {
03:    PexAssume.IsTrue(sn.Length == sv.Length);
04:    PexAssume.IsTrue(sn.Length > 0);
05:    MSS mss = new MSS();
06:    for(int count=0;count<sn.Length;count++){
07:        mss.SaveSetting(sn[count], sv[count]);
08:    }
09:    return mss;
}
```

Fig. 6. An example factory method for the `MemorySettingsStorage` class

³ `PexAssumeUnderTest` is a custom attribute provided by Pex, shown as “PAUT” for simplicity in Figures 5, 6, and 9.

Table 1.

(a) Names.		(b) Characteristics of subject applications.					(c) Existing CUTs.		
Subject Applications	Downloads	Code Under Test					#CUTs	Test Code #KLOC	
		#C	#M	#KLOC	Avg.CC	Max.CC			
NUnit	193,563	9	87	1.4	1.48	14.0	9	49	0.9
DSA	3239	27	259	2.4	2.09	16.0	20	337	2.5
QuickGraph	7969	56	463	6.2	1.79	16.0	9	21	1.2

help reduce effort required for this step. However, since Pex does not include these techniques yet, the developer can assist Pex by writing method-call sequences inside factory methods, supported by Pex. Figure 6 shows an example factory method for the `MemorySettingsStorage` class.

S5 - Add Mock Object. Pex (or any other existing DSE-based approaches) also faces challenges in handling PUTs or MT that interacts with an external environment such as a file system. To address this challenge related to the interactions with the environment, developers write mock objects for assisting Pex [16]. These mock objects help test features in isolation especially when PUTs or MT interact with environments such as a file system. Recent work [16] on mock objects can further help reduce effort in writing mock objects.

Generalized PUT. Figure 5 shows the final PUT after the developer follows our methodology. The PUT accepts three parameters: an instance of `SettingsGroup`, the name of the setting, and its value. The `SaveSetting` method can be used to save either an `int` value or a `string` value (the method accepts both types for its arguments). Therefore, the CUT requires two method calls shown in Statements 3 and 4 of Figure 4 to verify whether `SaveSetting` correctly handles these types. On the other hand, only one method call is sufficient in the PUT, since the two constant setting values are promoted to a PUT parameter of type `object`. Pex automatically explores the MT and generates CUTs that cover both `int` and `string` types. Indeed, the `SaveSetting` method also accepts `bool` and `enum` types. The existing CUTs did not include test data for verifying these two types. Our generalized PUT automatically handles these additional types, highlighting additional advantage of test generalization in reducing the test code substantially without reducing the behavior exercised by existing CUTs. When we applied Pex on the PUT shown in Figure 5, Pex generated 8 CUTs from the PUT. These CUTs test the `SaveSetting` method with different setting values of types such as `int`, `string`, or other non-primitive object types. Furthermore, the CUT used for generalization achieved branch coverage of 10%, whereas the CUTs generated from the generalized PUT achieved branch coverage of 90%, showing the benefits achieved through our test generalization methodology.

4 Empirical Study

We conducted an empirical study using three real-world applications to show the benefits of retrofitting CUTs as PUTs. In our empirical study, we show the cost and benefits

of PUTs over existing CUTs using *three metrics*: branch coverage, the number of detected defects, and the time taken for test generalization. In particular, we address the following three research questions in our empirical study:

- **RQ1: Branch Coverage.** How much higher percentage of *branch coverage* is achieved by retrofitted PUTs compared to existing CUTs? Since PUTs are a generalized form of CUTs, this research question helps address whether PUTs can achieve additional branch coverage compared to CUTs. We focus on branch coverage, since detecting defects via violating test assertions in unit tests can be mapped to covering implicit checking branches for those test assertions.
- **RQ2: Defect Detection.** How many new *defects* (that are not detected by existing CUTs) are detected by PUTs and vice-versa? This research question helps address whether PUTs have higher fault-detection capabilities compared to CUTs.
- **RQ3: Generalization Effort.** How much effort is required for generalizing CUTs to PUTs? This research question helps show that the effort required for generalization is worthwhile, considering the generalization benefits.

We first present the details of subject applications and next describe our setup for our empirical study. Finally, we present the results of our empirical study. The detailed results of our empirical study are available at our project website <https://sites.google.com/site/asergroup/projects/putstudy>.

4.1 Subject Applications

We use three popular open source applications (as shown by their download counts in their hosting web sites) in our study: NUnit [7], DSA [11], and Quickgraph [12]. Table 1(a) shows the names of three subject applications. While we used all namespaces and classes for DSA and QuickGraph in our study, for NUnit, we used nine classes from its `Util` namespace, which is one of the core components of the framework. Table 1(b) shows the characteristics of the three subject applications. Column “Downloads” shows the number of downloads of the application (as listed in its hosting web site in January 2011). Column “Code Under Test” shows details of the code under test (of the application) in terms of the number of classes (“#C”), number of methods (“#M”), number of lines of code (“#KLOC”), and the average and maximum cyclometric complexity (“Avg.CC” and “Max.CC”, respectively) of the code under test. Similarly, Table 1(c) shows the statistics of existing CUTs for these subject applications.

4.2 Study Setup

We next describe the setup of our study conducted by the first and second authors of this paper for addressing the preceding research questions. The authors were PhD (fourth year) and master (second year) students, respectively, with the same experience of two years with PUTs and Pex at the time of conducting the study. Before joining their graduate program, the authors had three and five years of programming experience, respectively, in software industry. Each of the authors conducted test generalization for half of CUTs across all the three subjects. The authors do not have prior knowledge of the

Table 2. Branch coverage achieved by the existing CUTs, CUTs + RTs, and CUTs generated by Pex using the retrofitted PUTs

Subject	Branch Coverage			Overall Inc.	Max. Inc.
	CUTs	CUTs+RTs(#)	PUTs		
JUnit	78%	78%(144)	88%	10%	52%
DSA	91%	91%(615)	92%	1%	1%
QuickGraph	87%	88%(3628)	89%	2%	11%

subject applications and conducted the study as third-party testers. We expect that our test-generalization results could be much better, if the test generalization is performed by the developers of these subject applications. The reason is that these developers can incorporate their application knowledge during test generalization to write more effective PUTs.

To address the preceding research questions, the authors used three categories of CUTs. The first category of CUTs is the set of existing CUTs available with subject applications. The second category of CUTs is the set of CUTs generated from PUTs. To generate this second category of CUTs, the authors generalized existing CUTs to PUTs and applied Pex on those PUTs. Among all three subject applications, the authors retrofitted 407 CUTs (4.6 KLOC) as 224 PUTs (4.0 KLOC). The authors also measured the time taken for generalizing all CUTs to compute the generalization effort for addressing RQ3. The measured time includes the amount of time taken for performing all steps described in our methodology and also applying Pex to generate CUTs from PUTs. The authors wrote 10 factory methods and 1 mock object during test generalization. The third category of CUTs is the set of existing CUTs + new CUTs (hereby referred to as *RTs*) that were generated using an automatic random test-generation tool, called Randoop [18]. The authors used the default timeout parameter of 180 seconds. The rationale behind using the default timeout is that running Randoop for longer time often generates a large number of tests that are difficult to be compiled. This third category (CUTs + RTs) helps show that the benefits of test generalization cannot be achieved by simply generating additional tests using tools such as Randoop. To address RQ1, the authors measured branch coverage using a coverage measurement tool, called NCover⁴. To address RQ2 and RQ3, the authors measured the number of failing tests and computed the code metrics (LOC) using the CLOC⁵ tool, respectively. The authors did not compare the execution time of CUTs for all three categories, since the time taken for executing CUTs of all categories is negligible (< 20 sec).

4.3 RQ1: Branch Coverage

We next describe our empirical results for addressing RQ1. Table 2 shows the branch coverage achieved by executing the existing CUTs, CUTs + RTs, and the CUTs generated by Pex using the retrofitted PUTs. The values in brackets (#) for CUTs + RTs

⁴ <http://www.ncover.com/>

⁵ <http://cloc.sourceforge.net/>

indicate the number of RTs, i.e., the tests generated by Randoop. Column “Overall Inc.” shows the overall increase in the branch coverage from the existing CUTs to the retrofitted PUTs. Column “Max. Inc.” shows the maximum increase for a class or namespace in the respective subject applications.

Column “Overall Inc.” shows that the branch coverage is increased by 10%, 1%, and 2% for NUnit, DSA, and QuickGraph, respectively. Furthermore, Column “Max Inc.” shows that the maximum branch coverage for a class or a namespace is increased by 52%, 1%, and 11% for NUnit, DSA, and QuickGraph, respectively. One major reason for not

achieving an increase in the coverage for DSA is that the existing CUTs already achieved high branch coverage and PUTs help achieve only a little higher coverage than existing CUTs.

To show that the increase in the branch coverage achieved by PUTs is not trivial to achieve, we compare the results of PUTs with CUTs + RTs. The increase in the branch coverage achieved by CUTs + RTs compared to CUTs alone is 0%, 0%, and 1% for NUnit, DSA, and QuickGraph, respectively. This comparison shows that the improvement in the branch coverage achieved by PUTs is not trivial to achieve, since the branches that are not covered by the existing CUTs are generally quite difficult to cover (as shown in the results of CUTs + RTs).

4.4 RQ2: Defects

To address RQ2, we identify the number of defects detected by PUTs. We did not find any failing CUTs among existing CUTs of the subject applications. Therefore, we consider the defects detected by failing tests among the CUTs generated from PUTs as new defects not detected by existing CUTs. In addition to the defects detected by PUTs, we also inspect the failing tests among the RTs to compare the fault-detection capabilities of PUTs and RTs.

In summary, our PUTs found 15 new defects in DSA and 4 new defects in NUnit. After our inspection, we reported the failing tests on their hosting websites⁶. On the other hand, RTs

```
01:public void RemoveSetting(string sn) {
02: int dot = settingName.IndexOf('.');
03: if (dot < 0)
04:     key.DeleteValue(settingName, false);
05: else {
06:     using(RegistryKey subKey = key.OpenSubKey(
07:         sn.Substring(0,dot),true)) {
08:         if (subKey != null)
09:             subKey.DeleteValue(sn.Substring(dot+1));
09:     }
09: }
```

Fig. 7. RemoveSetting method whose coverage is increased by 60% due to test generalization

```
//To test Remove item not present
01:public void RemoveCUT() {
02: Heap<int> actual = new Heap<int>{
03:     2, 78, 1, 0, 56};
03: Assert.IsFalse(actual.Remove(99));
03: }
```

Fig. 8. Existing CUT to test the Remove method of Heap

```
01:public void RemoveItemPUT (
02: [PAUT]List<int> in, int item) {
03: Heap<int> ac = new Heap<int>(in);
03: if (input.Contains(item)) {
04:     ....
05: } else {
06:     PexAssert.IsFalse(ac.Remove(randomPick));
07:     PexAssert.AreEqual(in.Count, ac.Count);
08:     CollectionAssert.AreEquivalent(ac, in);
09: }
```

Fig. 9. A generalized PUT of the CUT shown in Fig. 8

⁶ Reported bugs can be found at the DSA CodePlex website with defect IDs from 8846 to 8858 and the NUnit SourceForge website with defect IDs 2872749, 2872752, and 2872753.

include 90, 25, and 738 failing tests for DSA, NUnit, and QuickGraph, respectively. Since RTs are generated automatically using Randoop, RTs do not include test oracles. Therefore, an RT is considered as a failing test, if the execution of the RT results in an uncaught exception being thrown. In our inspection of these failing tests in RTs, we found that only 18 failing tests for DSA are related to 4 real defects in DSA, since the same defect is detected by multiple failing tests. These 4 defects are also detected by our PUTs. The remaining failing tests are due to two major issues. First, exceptions raised by RTs are expected. In our methodology, we address this issue by adding annotations to PUTs regarding expected exceptions. We add these additional annotations based on expected exceptions in CUTs. Second, illegal test data such as `null` values are passed as arguments to methods invoked in RTs. In our methodology, we address this issue of illegal test data by adding assumptions to PUTs in Step S1. This issue of illegal test data in RTs shows the significance of Step S1 in our methodology.

To further show the significance of generalized PUTs, we applied Pex on these applications without using these PUTs and by using *PexWizard*. *PexWizard* is a tool provided with Pex and this tool automatically generates PUTs (without test oracles) for each public method in the application under test. We found that the generated CUTs include 23, 170, and 17 failing tests for DSA, NUnit, and QuickGraph, respectively. However, similar to Randoop, only 2 tests are related to 2 real defects (also detected by our generalized PUTs) in DSA, and the remaining failing tests are due to the preceding two issues faced by Randoop.

We next explain an example defect detected in the `Heap` class of the DSA application by CUTs generated from generalized PUTs. The details of remaining defects can be found at our project website. The `Heap` class is a heap implementation in the `DataStructure` namespace. This class includes methods to add, remove, and heapify the elements in the heap. The `Remove` method of the class takes an item to be removed as a parameter and returns `true` when the item to be removed is in the heap, and returns `false` otherwise. Figure 8 shows the existing CUT that checks whether the `Remove` method returns `false` when an item that is not in the heap is passed as the parameter. On execution, this CUT passed, exposing no defect in the code under test, and there are no other CUTs (in the existing test suite) that exercise the behavior of the method. However, from our generalized PUT shown in Figure 9, a few of the generated CUTs failed, exposing a defect in the `Remove` method. The test data for the failing tests had the following common characteristics: the heap size is less than 4 (the `input` parameter of the PUT is of size less than 4), the item to be removed is 0 (the `item` parameter of the PUT), and the item 0 was not already added to the heap (the generated value for `input` did not contain the item 0).

When we inspected the causes of the failing tests, we found that in the constructor of the `Heap` class, a default array of size 4 (of type `int`) is created to store the items. In C#, an integer array is by default assigned values zero to the elements of the array. Therefore, there is always an item 0 in the heap unless an input list of size greater than or equal to 4 is passed as the parameter. Therefore, on calling the `Remove` method to remove the item 0, even when there is no such item in the heap, the method returns `true` indicating that the item has been successfully removed and causing the assertion statement to fail (Statement 6 of the PUT). However, this defect was not detected by the

CUT shown in Figure 8 since the unit test assigns the heap with 5 elements (Statement 2) and therefore the defect-exposing scenario of heap size ≤ 4 is not exercised. These 19 new defects that were not detected by the existing CUTs show that PUTs are an effective means for rigorous testing of the code under test. Furthermore, as described earlier, it is also difficult to write new CUTs (manually) that test corner cases as exercised by CUTs generated from PUTs.

4.5 RQ3: Generalization Effort

We next address RQ3 regarding the manual effort required for the generalization of CUTs to PUTs. The first two authors conducted comparable amount of generalization by equally splitting the existing CUTs of all three subject applications for generalization. The cumulative effort of both the authors in conducting the study is 2.8, 13.8, and 1.5 hours for subject applications NUnit, DSA, and QuickGraph, respectively. Our measured timings are primarily dependent on four factors: the expertise with PUTs and the Pex tool, prior knowledge of the subject applications, number of CUTs and the number of generalized PUTs, and the complexity of a CUT or a generalized PUT. Although the authors have experience with PUTs and using Pex, the authors do not have the prior knowledge of these subject applications and conducted the study as third-party testers. Therefore, we expect that the developers of these subject applications, despite unfamiliar with PUTs or Pex, may take similar amount of effort. Overall, our results show that the effort of test generalization is worthwhile considering the benefits that can be gained through generalization.

5 Threats to Validity

The threats to external validity primarily include the degree to which the subject programs, defects, and CUTs are representative of true practice. The subject applications used in our empirical study range from small-scale to medium-scale applications that are widely used as shown by their number of downloads. We tried to alleviate the threats related to detected defects by inspecting the source code and by reporting the defects to the developers of the application under test. These threats could further be reduced by conducting more studies with wider types of subjects in our future work. The threats to internal validity are due to manual process involved in generalizing CUTs to PUTs and only two human subjects involved in the study. Our study results can be biased based on our experience and knowledge of the subject applications. These threats can be reduced by conducting more case studies with more subject applications and additional human subjects. The results in our study can also vary based on other factors such as test-generation capability of Pex.

6 Related Work

Pex [22] accepts PUTs and uses dynamic symbolic execution to generate test inputs. Although we use the Pex terminology in describing our generalization procedure, our

procedure is independent of Pex and can be applied with other testing tools that accept unit tests with parameters such as JUnitFactory [1] for Java testing. Other existing tools such as Parasoft Jtest [2] and CodeProAnalytiX [3] adopt the design-by-contract approach [17] and allow developers to specify method preconditions, postconditions, and class invariants for the unit under test and carry out symbolic execution or random testing to generate test inputs. More recently, Saff et al. [19] propose theory-based testing and generalize six Java applications to show that the proposed theory-based testing is more effective compared to traditional example-based testing. A theory is a partial specification of a program behavior and is a generic form of unit tests where assertions should hold for all inputs that satisfy the assumptions specified in the unit tests. A theory is similar to a PUT and Saff et al.'s approach uses these defined theories and applies the constraint solving mechanism based on path coverage to generate test inputs similar to Pex. In contrast to our study, their study does not provide a systematic procedure of writing generalized PUTs or show empirical evidence of benefits of PUTs as shown in our study.

There are existing approaches [8, 18, 14] that automatically generate required method-call sequences that achieve different object states. However, in practice, each approach has its own limitations. For example, Pacheco et al.'s approach [18] generates method-call sequences randomly by incorporating feedback from already generated method-call sequences. However, such a random approach can still face challenges in generating desirable method-call sequences, since often there is little chance of generating required sequences at random. In our test generalization, we manually write factory methods to assist Pex in generating desirable object states for non-primitive data types, when Pex's existing sequence-generation strategy faces challenges.

In our previous work [16], we presented an empirical study to analyze the use of parameterized mock objects in unit testing with PUTs. We showed that using a mock object can ease the process of unit testing and identified challenges faced in testing code when there are multiple APIs that need to be mocked. In our current study, we also use mock objects in our testing with PUTs. However, our previous study showed the benefits of mock objects in unit testing, while our current study shows the use of mock objects to help achieve test generalization. In our other previous work with PUTs [25], we propose mutation analysis to help developers in identifying likely locations in PUTs that can be improved to make more general PUTs. In contrast, our current study suggests a systematic procedure of retrofitting CUTs for parameterized unit testing.

7 Conclusion

Recent advances in software testing introduced parameterized unit tests (PUTs) [23], which are a generalized form of conventional unit tests (CUTs). With PUTs, developers do not need to provide test data (for PUTs), which are generated automatically using state-of-the-art test-generation approaches such as dynamic symbolic execution. Since many existing applications often include manually written CUTs, in this paper, we present an empirical study to investigate whether existing CUTs can be retrofitted as PUTs to leverage the benefits of PUTs. We also proposed a methodology, called test generalization, for systematically retrofitting CUTs as PUTs. Our empirical results

show that test generalization helped detect 19 new defects and also helped achieve additional branch coverage of the code under test. In future work, we plan to automate our methodology to further reduce the manual effort required for test generalization. Furthermore, given the results of our current study, we plan to conduct further empirical study to compare the cost and benefits involved in writing PUTs directly, and writing CUTs first and generalizing those CUTs as PUTs using our methodology.

Acknowledgments

This work is supported in part by NSF grants CCF-0725190, CCF-0845272, CCF-0915400, CNS-0958235, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU SOSI.

References

1. Agitar JUnit Factory (2008),
http://www.agitar.com/developers/junit_factory.html
2. Parasoft Jtest (2008),
<http://www.parasoft.com/jsp/products/home.jsp?product=Jtest>
3. CodePro AnalytiX (2009), http://www.eclipse-plugins.info/eclipse/plugin_details.jsp?id=943
4. Pex - automated white box testing for .NET (2009),
<http://research.microsoft.com/Pex/>
5. Code Contracts (2010), <http://research.microsoft.com/en-us/projects/contracts/>
6. Pex Documentation (2010),
<http://research.microsoft.com/Pex/documentation.aspx>
7. Cansdale, J., Feldman, G., Poole, C., Two, M.: NUnit (2002),
<http://nunit.com/index.php>
8. Csallner, C., Smaragdakis, Y.: JCrasher: an automatic robustness tester for Java. *Softw. Pract. Exper.* 34(11) (2004)
9. Daniel, B., Jagannath, V., Dig, D., Marinov, D.: ReAssert: Suggesting repairs for broken unit tests. In: Proc. ASE, pp. 433–444 (2009)
10. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Proc. PLDI, pp. 213–223 (2005)
11. Granville, B., Tongo, L.D.: Data structures and algorithms (2006),
<http://dsa.codeplex.com/>
12. de Halleux, J.: Quickgraph, graph data structures and algorithms for .NET (2006),
<http://quickgraph.codeplex.com/>
13. Jaygarl, H., Kim, S., Xie, T., Chang, C.K.: OCAT: Object capture-based automated testing. In: Proc. ISSTA, pp. 159–170 (2010)
14. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003)
15. King, J.C.: Symbolic execution and program testing. *Communications of the ACM* 19(7), 385–394 (1976)

16. Marri, M.R., Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: An empirical study of testing file-system-dependent software with mock objects. In: Proc. AST, Business and Industry Case Studies, pp. 149–153 (2009)
17. Meyer, B.: Object-Oriented Software Construction. Prentice Hall PTR, Englewood Cliffs (2000)
18. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: Proc. ICSE, pp. 75–84 (2007)
19. Saff, D., Boshernitsan, M., Ernst, M.D.: Theories in practice: Easy-to-write specifications that catch bugs. Tech. Rep. MIT-CSAIL-TR-2008-002, MIT Computer Science and Artificial Intelligence Laboratory (2008), <http://www.cs.washington.edu/homes/mernst/pubs/testing-theories-tr002-abstract.html>
20. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proc. ESEC/FSE, pp. 263–272 (2005)
21. Thummalapenta, S., Xie, T., Tillmann, N., de Halleux, P., Schulte, W.: MSeqGen: Object-oriented unit-test generation via mining source code. In: Proc. ESEC/FSE, pp. 193–202 (2009)
22. Tillmann, N., de Halleux, J.: Pex—white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
23. Tillmann, N., Schulte, W.: Parameterized Unit Tests. In: Proc. ESEC/FSE, pp. 253–262 (2005)
24. Xie, T., Marinov, D., Notkin, D.: Rostra: A framework for detecting redundant object-oriented unit tests. In: Proc. ASE, pp. 196–205 (2004)
25. Xie, T., Tillmann, N., de Halleux, P., Schulte, W.: Mutation analysis of parameterized unit tests. In: Proc. Mutation, pp. 177–181 (2009)

Evolving a Test Oracle in Black-Box Testing

Farn Wang^{1,2}, Jung-Hsuan Wu¹, Chung-Hao Huang^{1,2}, and Kai-Hsiang Chang¹

¹ Dept. of Electrical Engineering, ² Graduate Institute of Electronic Engineering
National Taiwan University, Taiwan, ROC
farn@cc.ee.ntu.edu.tw

Abstract. Software testing is an important and expensive activity to the software industry, with testing accounting for over 50% of the cost of software. To ease this problem, test automation is very critical to the process of software testing. One important issue in this automation is to automatically determine whether a program under test (PUT) responds the correct(expected) output for an arbitrary input. In this paper, we model PUTs in black-box way, i.e. processing and responding a list of numbers, and design input/output list relation language(IOLRL) to formally describe the relations between the input and output lists. Given several labelled test cases(test verdicts are set), we use genetic programming to evolve the most distinguishing relations of these test cases in IOLRL and encode the test cases into bit patterns to build a classifier with support vector machine as the constructed test oracle. This classifier can be used to automatically verify if a program output list is the expected one in processing a program input list. The main contribution of this work are the designed IOLRL and the approach to construct test oracle with evolve relations in IOLRL. The experiments show the constructed test oracle has good performance even when few labelled test cases are supplied.

Keywords: test oracle, input/output list relation language, genetic programming, support vector machine, black-box testing.

1 Introduction

Software testing is an important and expensive activity to the software industry, with testing accounting for over 50% of the cost of software [6]. To ease this situation, test automation is very critical to the process of software testing [4,5]. One important issue in the test automation is an automated test oracle. A test oracle is a mechanism to determine whether an output is the expected output of a program under test (PUT) against an input. Once the test oracle can be automated, the correctness of program outputs can be verified without human intervention. Therefore the efficiency of software testing process can be enhanced.

In this paper, we investigate the problem of test oracle automation in black-box approach. The behavior of a PUT is modelled to process a list of numbers and respond a list of numbers. We design input/output list relation language(IOLRL) to describe the relations between the input and output lists and propose an

approach to construct a test oracle with labelled test cases by incorporating the techniques of genetic programming and support vector machine. With genetic programming, relations in IOLRL are evolved to describe the behaviors of a PUT and used to encode the labelled test cases into bit patterns. With these bit patterns, an SVM classifier can be trained and used to verify if an output list is the expected one of a PUT in processing an input list. Hence an automated test oracle is constructed.

The remainder of this paper is structured as follows: Section 2 reviews the related work. Section 3 reviews the background materials of software testing, genetic programming and support vector machine. Section 4 introduces the input/output list relation language(IOLRL). Section 5 presents the procedures of our proposed approach. Section 6 reports the experiments. Section 7 is the conclusion.

2 Related Work

In the literature, there have been several researches addressing on the issue of test oracle automation. Brown, et al, [1] use white-box approach to automatically generate a test oracle from formally specified requirements. Peters, et al, [8] constructs a test oracle from program documentations. Chen, et al, [3] constructs a test oracle with metamorphic testing which needs to identify the PUT properties first. In contrast to these approaches, our approach is black-box based and constructs test oracle from test cases which are generally easier for users to supply.

The approach proposed by Vanmali, et al, [10] also construct a test oracle from test cases with neural network technique. Compare to their approach, our approach emphasizes on the relation of the input and output data of a PUT and automatically figures out the relations to infer the PUT's behavior with these data. The experiments in section 6 shows that our approach performs better with fewer supplied test cases. In some cases with large amount of supplied test cases, our approach can also performs better.

3 Background

In this section, we first introduce the model of a PUT in this paper and review the definitions of test oracle and test case. Then the background knowledge about genetic programming and support vector machine are introduced.

3.1 Definitions

The considered PUTs in this paper are abstracted into programs that take sequences of integers as their input and respond sequences of integers as their output. We also assume the behaviors of such PUTs can only be observed from the input lists and output lists. Therefore we have the following definitions. For convenience, we use \mathbb{N} to denote the non-negative numbers and \mathbb{Z} to denote the integer numbers.

Definition 1. (*Input list*) An input list $I : \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z})$ is a finite sequence such that each element of I is a finite sequence of integers. ■

Two input lists I, I' are equal if $I(i)(j) = I'(i)(j)$, $0 \leq i, j$.

Definition 2. (*Output list*) An output list $O : \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z})$ is a finite sequence such that each element of O is a finite sequence of integers. ■

Two output lists O, O' are equal if $O(i)(j) = O'(i)(j)$, $0 \leq i, j$.

Definition 3. (*PUT*) A program under test (*PUT*) $\pi : (\mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z})) \rightarrow (\mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z}))$ is a partial function that maps an input list to an output list. ■

Note that we restrict the size of sequences of input and output lists to be finite. This restriction is natural since computer systems are always restricted to finite sets in practice [8]. We use example 1 to explain the formal modelling of the considered PUT.

Example 1. Consider a list-processing program that searches for an integer out of a sequence of numbers. If the sequence contains this integer, the index of this integer in the sequence will be returned, otherwise -1 will be returned. With the above formal notation, let $I = \{\{3, 10, 36, 5\}, \{5\}\}$ be an input list of this program. If this program responds $O = \{\{3\}\}$, then $\{I \rightarrow O\}$ is one of the mappings of this program.

Definition 4. (*Test oracle*) A test oracle of a PUT is a total function $\omega : (\mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z}), \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z})) \rightarrow \{\text{pass}, \text{fail}\}$. Given an input list I and an output list O , $\omega(I, O) = \text{pass}$ if O is the correct output of this PUT in processing I . Otherwise $\omega(I, O) = \text{fail}$. Ideally, the mapping of a test oracle is always correct. ■

Intuitively, a test case is a description to describe the output list of a PUT in processing an input list. This test case is also labelled with a *verdict* bit to indicate whether the described output list is the correct response. We formally define a test case as follows.

Definition 5. (*Test case*) A test case $t = (I, O, v)$ is a triple where

- I is an input list,
- O is an output list, and
- $v \in \{\text{pass}, \text{fail}\}$ is a verdict bit to label this test case. $v = \text{pass}$ indicates O is a correct response against I . Otherwise O is not a correct response.

For convenience, we denote t^I the input list of t , t^O the output list of t and t^v the verdict bit of t . ■

For example, with the PUT illustrated in example 1, a test case $(\{\{3, 10, 36, 5\}, \{5\}\}, \{\{-1\}\}, \text{fail})$ describes “ -1 ” is not the expected output list in the searching for 5 out of the sequence $\{3, 10, 36, 5\}$. Another test case $(\{\{3, 10, 36, 5\}, \{5\}\}, \{\{3\}\}, \text{pass})$ describes “ 3 ” is expected in the searching for 5.

3.2 Genetic Programming

Genetic programming (GP) [7, 12, 9] is an evolutionary computation (EC) technique that automatically solves problems without requiring the user to know or specify the form or structure of the solution in advance. It follows the Darwin's theory that transforms populations of *individuals* into new populations of individuals generation by generation. GP, like nature, is a random process, and it can never guarantee results. GP's essential randomness, however, can lead it to escape traps which deterministic methods may be captured by. Like nature, GP has been very successful at evolving novel and unexpected ways of solving problems [9].

GP starts with a population of randomly generated individuals. Each individual is scored with a *fitness function*. Then, these individuals will *reproduce* itself, *crossover* with another individual and *mutate* into a new population. Individuals with higher scores will perform these operations more frequently. Then the average fitness value of the new evolved population will, almost certainly, be better than the average of the previous population. The individuals in the new population are evaluated again according to the fitness function. Operations such as reproduction, crossover and mutation are again applied to this new population. After sufficient generations, the best individual in the last population would be an excellent solution to the posed problem.

The key components of GP are as follows:

- A *syntax* to represent individuals. In GP, individuals are usually expressed as syntax trees.
- A *fitness function* to evaluate how well each individual can survive in the environment.
- A method to initialize a population. The *full*, *grow* and *ramped half-and-half* methods are common in this regard. In both the full and grow methods, the initial individuals, i.e. syntax trees, are generated so that they do not exceed a user specified maximum depth and the tree nodes are randomly taken. The difference is the syntax trees generated by the full method are complete and the syntax trees generated by the grow method are not necessary to be complete. In the ramped half-and-half method, it simply generates half of the population by the full method and half of the population by the grow method to help ensure the generated trees having a variety of sizes and shapes.
- A genetic operator to probabilistically select better individuals based on fitness. The most commonly employed method is *tournament selection*. In tournament selection, a number of individuals are chosen at random from the population. Then these chosen individuals are compared with each other and the top n are chosen to be the parents (n is decided according to the number of parents it needed in the successive genetic operator). Note that the number of randomly chosen individuals should not be too large in order to keep the diversity of individuals in each generation.
- Genetic operators to generate offspring individuals from the parent individuals. The common used operator are *subtree crossover*, *subtree mutation* and

reproduction. Given two parent individuals, subtree crossover randomly selects a node in each parent individual and swaps the entire subtrees. Given a parent individual, the subtree mutation randomly selects a node and substitutes the subtree rooted there with a randomly generated subtree. Given a parent individual, the reproduction operator simply inserts a copy of it into the next generation.

- A termination criterion to stop evolving new population. Usually the termination criterion is a number specified by user to determine the maximum generation to evolve.

Due to the page limit, we refer the interesting readers to [7, 12, 9] for the details of these components.

The syntax should have an important property call *Sufficiency* which means it is possible to express a solution to the problem using the syntax. Unfortunately, sufficiency can be guaranteed only for those problems where theory, or experience with other methods, tells us that a solution can be obtained with the syntax [9]. When a syntax is insufficiency, GP can only develop individuals that approximate the desired one. However, in many cases such an approximation can be very close and good enough for the user’s purpose.

For GP to work effectively, most non-terminal nodes in a syntax tree are also required to have another important *closure* property [7] which can be further broken down into the *type consistency* and *evaluation safety* properties. Since the crossover operator can mix and join syntax nodes arbitrarily, it is necessary that any subtree can be used in any of the argument positions for every non-terminal node. Therefore it is common to require all the non-terminal nodes to be type consistent, i.e. they all return values of the same type, and each of their arguments also have this type. An alternative to require the type consistency is to extend the crossover and mutation operators to explicitly make use of the type information so that the offsprings they produce do not contain illegal type mismatches. The other component of closure is evaluation safety. Evaluation safety is required because individuals may fail in an execution. In order to preserve the evaluation safety property, modifications of normal behaviors are common. An alternative is to reduce the fitness of individuals that have such errors. In section 5, we will describe how to handle the closure and evaluation safety properties in our application.

3.3 Support Vector Machine

The support vector machine (SVM) [11, 13] is a technique motivated by statistical learning theory and has been extensively used to solve many classification problems. It is a method to learn functions from a set of labelled training data points. The idea is to separate two classes of the labelled training data points with a decision boundary which maximizes the margin between them. In this paper, we will only consider a binary classification task with a set of n labelled training data points: (\mathbf{x}_i, y_i) , where $\mathbf{x}_i \in \mathbb{R}^N$ and $y_i \in \{+1, -1\}$ indicates its corresponding class. The SVM technique tries to separate the training data points

with a hyperplane $\mathbf{x}_i^T \mathbf{w} + b = 0$ with the maximum margin where $\mathbf{w} \in \mathbb{R}^N$ is normal to the hyperplane and $b \in \mathbb{R}$ is a bias. The *margin* of such hyperplane is defined by the sum of the shortest distance from the hyperplane to the closest positive training data points and the closest negative training data points. Due to the page limit, we refer the interesting readers to [11, 13] for the details.

One of the advantages of SVM is that the learning task is insensitive to the relative numbers of training data points in positive and negative classes. Another advantage is SVM can achieve a lower false alarm rate. Most learning algorithm based on Empirical Risk Minimization will tend to classify only the positive class correctly to minimize the error over data set. Since SVM aims at minimizing a bound on the generalization error of a model in high dimensional space, so called Structural Risk Minimization, rather than minimizing the error over data set, the training data points that are far behind the hyperplane will not change the support vectors. Therefore, SVM can achieve a lower false alarm rate.

4 Input/Output List Relation Language(IOLRL)

In this section, we present the *input/output list relation language* (IOLRL) to formally describe the relations between the input and output lists of test cases. The syntax of the language IOLRL in BNF form is as follows:

```

< Rule >   ::= < Rule >< BOP >< Rule >
              |  $\neg$  ( < Rule > )
              |  $\exists$  < qvar > ( < Rule > )
              |  $\forall$  < qvar > ( < Rule > )
              | < Elem >< ROP >< Elem >
              | < Elem >< InOP >< Elems >;
< Elems >  ::= < List > [ < nv > ];
< Elem >   ::= < Math > ( < List > [ < nv > ] )
              | < List > [ < nv > ][ < nv > ]
              | < nv > + < nv >
              | < nv > - < nv >
              | < nv >;
< Math >   ::= max | min | average | sum;
< BOP >   ::=  $\wedge$  |  $\vee$ ;
< ROP >   ::= = |  $\neq$  |  $\leq$  |  $>$  |  $<$  |  $\geq$ ;
< InOP >  ::=  $\in$  ;
< qvar >   ::= < var > ;
< nv >     ::= < num > | < var >;
< num >   ::= 0 | 1 | 2 | ... ;
< var >   ::= i | j | ... ;
< List >  ::= InputList | OutputList;

```

Note that for simplicity, the syntax rule $< num >$ is limited from 0 to 9 and the syntax rule $< var >$ is limited to i and j in the demonstration experiments in section 6.

The syntax of IOLRL are self-explained of its semantics. We use three example relations to explain the IOLRL semantics.

- $\exists i (\forall j (OutputList[0][j] \in InputList[i]))$. This relation describes there exists a sequence $InputList[i]$ such that every element of the sequence $OutputList[0]$ appears in $InputList[i]$.
- $\forall i (OutputList[0][i] \leq OutputList[0][i + 1])$. This relation describes the elements in the sequence $OutputList[0]$ are in non-decreasing order.
- $sum(InputList[0]) \in OutputList[0]$. This relation describes the summation of the sequence $InputList[0]$ is in the sequence $OutputList[0]$.

For convenience, we denote R the set of relations described in IOLRL. For a relation in R , we use it to test if the input/output lists of a test case satisfy this relation. Therefore we have the following definition.

Definition 6. (Evaluation) Given a test case t and a relation $r \in R$, an evaluation $\eta_r : (\mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z}), \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z})) \rightarrow \{0, 1\}$ is a function such that

$$\begin{cases} \eta_r(t^I, t^O) = 1 & \text{if } t \text{ satisfies } r \\ \eta_r(t^I, t^O) = 0 & \text{if } t \text{ does not satisfy } r \end{cases}$$
■

We introduce IOLRL and the evaluation in this paper for two main reasons.

1. We assume that an encoding with emphasis on the relation between the input and output lists can enhance the performance of the constructed test oracle.
2. Since we use SVM to construct the test oracle and the training data points processed by SVM are numbers, we need an encoding mechanism from a test case into numbers. The evaluation of test cases for each relation provides such transformation.

5 Implementation

This work is implemented with the auxiliaries of the genetic programming library *GPC++* [14] and the SVM library *libsvm* [2]. Given a set of labelled test cases as the training data and a PUT, the basic idea is to use GP to evolve several relations to well separate the *pass* and *fail* test cases. Here the individuals of GP are relations in IOLRL syntax. Then with the evolved relations, we train an SVM model based on the evaluations and labelling of these test cases. This trained SVM model is thus our constructed test oracle.

Figure 1 is the flowchart to construct a test oracle. We first explain the procedures as follows and the detail procedures in the next.

Step 1. Create an initial population with n relations by the ramped half-and half method. The maximum tree depth is arbitrarily set to 8.

Step 2. Evaluate each labelled test case with the n relations.

Step 3. Increase gen by 1. If gen does not exceed the user-defined maximum generation, calculate the fitness for each relation and go to step 4. Otherwise, go to step 5.

Step 4. Repeat to generate new offspring relations until n relations are generated. By setting the probability to 90% of performing crossover operation, Approximately 90% of the new population are generated with crossover operator. The rest of the population are generated with reproduction operator. Go to step 2. Note that 90% of crossover rate is set here because this is a typical setting [9]. We also simply eliminate the mutation operator because the probability to apply a mutation operator is relatively small (typically 1% rate).

Step 5. Train an SVM model as the constructed test oracle according to the evaluation of the training data. The used SVM kernel is radial basis function (RBF).

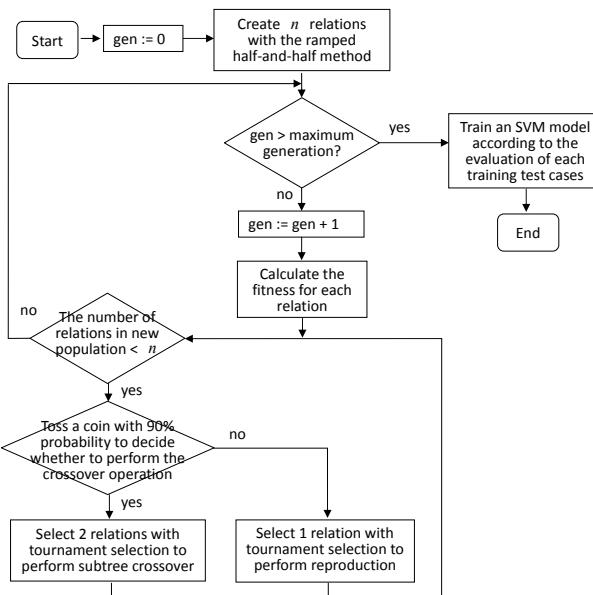


Fig. 1. The flowchart of the proposed method

Intuitively, a good relation should be able to distinguish test cases that are labelled to different verdicts and should not be able to distinguish test cases that are labelled to the same verdict. Therefore for those test cases that are labelled to *pass*, all the evaluations of these test cases should be either 0 or 1. And for those test cases that are labelled to *fail*, all the evaluations should be in the opposite side. The procedure to calculate the fitness of a relation in step 3 is described as follows. Note that the less returned number of this procedure means the better of the relation.

relation_fitness**input** A list of test cases T and a relation r .**output** A real number.

- 1: Let m and v be the mean and variance of $\{\eta_r(t^I, t^O) | t^v = \text{pass} \text{ and } t \in T\}$.
 - 2: Let m' and v' be the mean and variance of $\{\eta_r(t^I, t^O) | t^v = \text{fail} \text{ and } t \in T\}$.
 - 3: Return $v + v' - |m - m'|$.
-

To preserve the type consistency property in the generated relations, we implement the GP as *strongly typed GP* [9]. Therefore in an IOLRL syntax tree, every terminal symbol has a type and every non-terminal symbol has types for each of its arguments and a type for its return value. The process that generated the initial random individuals and the crossover operator are implemented under the type constraints. In our experiment, each left-hand-side symbol of IOLRL grammar rules has its own type. The procedure to create an individual in step 1 can only randomly pick a symbol of the demanded type to expand a syntax tree. The crossover operation in step 4 can only swap subtrees that return the same type.

As for the evaluation safety property, we can see that a generated relation may not be able to be evaluated. For example, $\forall i (\exists i (\text{OutputList}[0][j] < 9))$ is unable to be evaluated for any test case. Therefore for those relations that are unable to be evaluated, their fitness are set to the worst value among the population.

Since we construct a test oracle with SVM in step 5 and the input of SVM is a set of labelled numeric points, we need to prepare an input for SVM from the evolved relations and training data.bbvv. Therefore we have the following procedure. Then the training procedure of SVM is inferred to [2].

prepare_training_data**input** A list of relations R and a list of test cases T .**output** a set of training data points.

- 1: Let D be an empty set.
 - 2: **for** $i = 1$ to $|T|$ **do**
 - 3: Let t be the i th relation of T .
 - 4: Let M be a zero vector of size $|R|$.
 - 5: **for** $j = 1$ to $|R|$ **do**
 - 6: Let r be the j th relation of R .
 - 7: $m_j = \eta_r(t^I, t^O)$. /* m_j is the j th element of M . */
 - 8: **end for**
 - 9: **if** t^v is *pass* **then** $label = 1$ **else** $label = -1$ **end if**
 - 10: $D = D \cup (M, label)$.
 - 11: **end for**
 - 12: Return D .
-

6 Experiment

We use the following three benchmarks(PUTs) to demonstrate our technique.

- **Binary search.** The binary search program takes a numeric sequence and a number as its input and should respond the index of the number in the sequence. If the number is not in the sequence, -1 is responded. An example test case is ($\{\{1, 4, 5, 6, 7, 7\}, \{0\}\}, \{\{-1\}\}, pass$).
- **Quick sort.** The quick sort program takes a numeric sequence as its input and should respond the sequence in non-decreasing order. An example test case of this program is ($\{\{4, 5, 6, 8, 9, 4, 7, 10, 21\}\}, \{\{4, 4, 5, 6, 7, 8, 9, 10, 21\}\}, pass$).
- **Set intersection.** The set intersection program takes 2 numeric sequences as its input and should respond a sequence of the intersected numbers. In order to perform the operation efficiently, set intersection programs generally require the input sequences to be ordered and therefore the intersected sequence is also ordered. The benchmark here requires that the input and output sequences are in non-decreasing order. An example test case is ($\{\{1, 5, 7, 8, 9\}, \{1, 5, 9\}\}, \{\{1, 5, 9\}\}, pass$).

For each benchmark, we generate a data set with the following strategies.

- The half of the data set are test cases with *pass* verdict and the others are with *fail* verdict.
- For each test case, the numbers in the input list are generated at random between 1 to 50 and the size of the sequences in the input list are decided at random between 1 to 20.
- For test cases with *pass* verdict, the output lists are computed according to the correct behavior of the benchmark.
- For test cases with *fail* verdict, the output lists are generated with the following types of fault at uniform distribution.
 - **Binary search.** The output can be a random number other than the index of the searched number, the searched number itself rather than the index, and a sequence of randomly generated numbers.
 - **Quick sort.** The output can be a sequence of randomly generated numbers with different/the same size of the sequence in the input list, and an ordered sequence of randomly generated numbers other than the correct sequence.
 - **Set intersection.** The output can be a sequence of randomly generated numbers with random size/minimal size of sequences in the input list, and an unordered sequence of intersected numbers.

Note that the domain of randomly generated numbers and the size of sequences are limited for simplicity.

The experiments are conducted in two parts. The first part presents the performance data of our approach with different parameters and the next part presents the comparison data of the neural network approach in [10] and our approach with GP evolved/user specified relations.

6.1 Performance Study

Since our approach only require a set of labelled test cases to construct a test oracle, users may want to know how many test cases are sufficient and how to

setup the GP parameters with better performance. In this section, we present the experimental studies of training data size and GP population size. The performance of constructed test oracle is measured with *prediction accuracy* and *time cost*. The prediction accuracy is the percentage of correctly labelled test cases of a testing data set with the constructed test oracle. The time cost is the used time to evolve the relations and train the test oracle. The experimental data are collected on Intel Pentium CPU T4200@2.00 GHz with 3G RAM, running Ubuntu 8.04.

To objectively demonstrate the performance with different population size, we have collected the performance data for each benchmark with 4 different training data sets of size 50. For each benchmark, the prediction accuracy is tested with a testing data set of size 3000. Each experiment is run for 10 times and the average performance data is reported. As we can see in Figure 2, Figure 3 and Figure 4, the prediction accuracy climbs fast and is better with larger population size. The impact of population size is significant because greater diversity in the population can result in higher chance to evolve good relations. The time cost is summarized in Table 1, Table 2 and Table 3. The tables indicate our approach constructs a test oracle in reasonable time with good accuracy.

To objectively demonstrate the effect of different training data size, we have collected the performance data for each benchmark with 4 different populations. Each population consists of 400 relations and the maximal generation is 50. For each benchmark, the prediction accuracy is tested with a testing data set of 3000

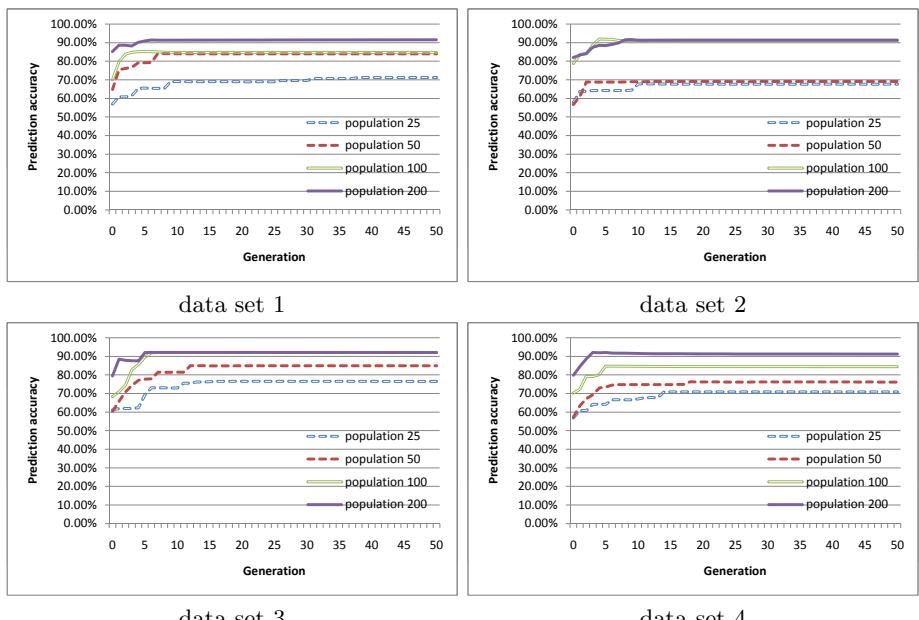


Fig. 2. Prediction accuracy with different population size for binary search benchmark

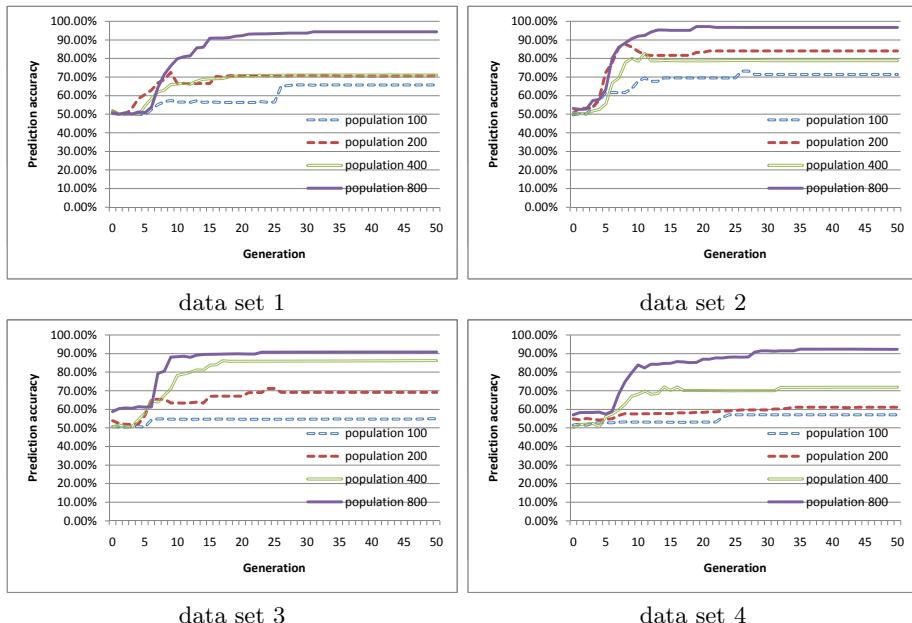


Fig. 3. Prediction accuracy with different population size for quick sort benchmark

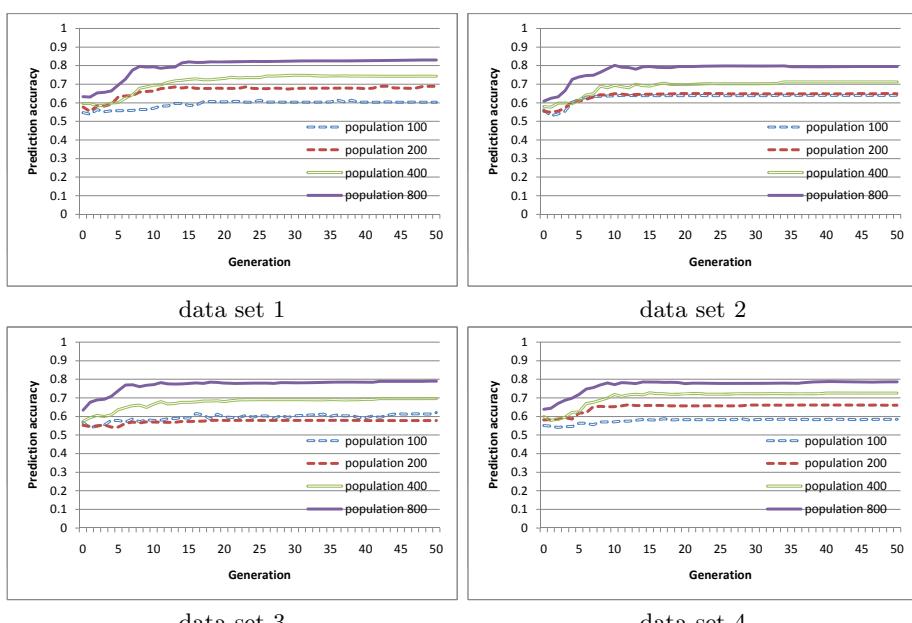


Fig. 4. Prediction accuracy with different population size for set intersection benchmark

Table 1. Performance data of binary search benchmark

Population size	25				50				100				200			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
accuracy(%)	71.1	67.6	76.5	70.8	83.9	69.1	84.9	76.2	84.7	91.2	92.0	84.4	91.6	91.4	92.1	91.2
time cost(s)	0.1	0.1	0.1	0.1	0.1	0.2	0.2	0.3	1.4	0.6	1.0	0.8	3.2	1.9	3.5	2.6

Table 2. Performance data of quick sort benchmark

Population size	100				200				400				800			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
accuracy(%)	65.7	71.4	54.9	57.1	70.8	84.0	69.1	61.1	71.1	78.7	86.1	71.8	94.3	96.7	90.9	92.3
time cost(s)	0.5	2.2	1.4	4.4	3.1	4.4	1.7	4.2	7.9	13.2	7.2	21.6	33.5	41.9	64.3	43.9

Table 3. Performance data of set intersection benchmark

Population size	100				200				400				800			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
accuracy(%)	60.3	64.0	62.0	58.4	68.9	64.8	57.7	66.1	74.3	71.2	69.5	72.5	83.0	79.5	79.0	78.6
time cost(s)	7.9	8.5	2.1	2.1	5.5	10.3	5.3	19.1	43.6	67.3	18.1	20.7	138.6	183.3	58.9	287.5

test cases. Each experiment is run for 10 times and the average performance data is reported. As we can see in Table 4, the training data size does not effect the prediction accuracy very much.

The parameter studies of the population size and training data size have demonstrated the key point of our approach is to have well-evolved relations to describe the behaviors of benchmarks rather than lots of training data. This is especially useful in applications that are lack of training data.

6.2 Compare with Manually Specified Relations and Neural Network Approaches

In the perfect condition of our approach, the GP evolved relations can completely distinguish the test cases of *pass* and *fail* verdicts. Therefore we show the prediction accuracy of our approach by replacing the GP evolved relations with manually specified relations in IOLRL. The manually specified relations describe the expected behaviors of the three benchmarks. For those test cases that are labelled *pass*, they will be evaluated to be *true* with these relations, and vice versa. Therefore, they can completely distinguish the *pass* and *fail* test cases. This experiment is to show the sufficiency of IOLRL as the input/output list relation specification language for the three benchmarks and the performance of the constructed test oracle by SVM. We also compare the performance data to our original approach to show the effectiveness of the GP procedures. At last, we compare with the neural network based approach proposed by [10].

Table 4. Prediction accuracy with different training data size

Benchmark	population	Training data size							
		10	20	50	100	200	400	800	1600
Binary search	1	86.1	89.7	79.1	87.8	92.1	91.5	91.5	91.7
	2	90.8	91.9	91.0	92.1	91.6	92.3	92.2	91.9
	3	91.8	91.8	92.2	91.9	92.3	92.0	92.0	92.3
	4	89.0	89.8	91.8	91.7	91.8	91.9	91.9	92.0
Quick sort	1	60.0	56.7	63.3	63.7	60.8	82.2	69.5	65.5
	2	63.5	61.2	59.1	61.9	57.5	58.7	75.4	70.7
	3	59.3	64.2	52.3	63.7	62.4	61.4	83.2	73.9
	4	62.9	66.2	68.4	55.6	68.9	59.9	65.4	69.2
Set intersection	1	54.7	61.3	54.8	56.7	58.9	56.7	57.6	58.9
	2	53.9	58.3	57.0	59.9	64.3	57.9	59.2	61.8
	3	57.8	53.6	57.1	65.3	59.9	58.2	53.3	64.1
	4	53.6	59.1	52.8	57.2	55.3	57.5	57.5	54.0

Table 5. Comparison of prediction accuracy with 50 training data

Benchmark	data set	Our approach	Our approach with manually specified relations	NN approach	NN approach with 400 training data
Binary search	1	91.6	99.90	77.13	95.47
	2	91.4		79.71	94.35
	3	92.1		75.39	94.72
	4	91.2		75.85	94.80
Quick sort	1	94.3	99.90	58.98	65.57
	2	96.7		56.23	66.33
	3	90.9		55.32	64.86
	4	92.3		56.00	67.01
Set intersection	1	83.0	98.44	75.93	86.19
	2	79.0		79.5	84.63
	3	79.5		79.0	86.51
	4	83.0		78.6	88.14

The manually specified relations are listed as follows.

- Binary search

$$\begin{aligned} & \forall i (InputList[0][i] \leq InputList[0][i + 1]) \\ & \wedge (OutputList[0][0] < 0) \vee (InputList[0][OutputList[0][0]] = InputList[1][0]) \\ & \wedge (OutputList[0][0] \geq 0) \vee (\neg (InputList[1][0] \in InputList[0])) \end{aligned}$$

- Quick sort

$$\begin{aligned} & \forall i (OutputList[0][i] \leq OutputList[0][i + 1]) \\ & \wedge \forall i (InputList[0][i] \in OutputList[0]) \\ & \wedge \forall i (OutputList[0][i] \in InputList[0]) \end{aligned}$$

– Set intersection

$$\begin{aligned}
 & \forall i (\text{OutputList}[0][i] \in \text{InputList}[0] \wedge \text{OutputList}[0][i] \in \text{InputList}[1]) \\
 & \wedge \forall i (\text{InputList}[0][i] \in \text{OutputList}[0] \vee \neg(\text{InputList}[0][i] \in \text{InputList}[1])) \\
 & \wedge \forall i (\text{InputList}[1][i] \in \text{OutputList}[0] \vee \neg(\text{InputList}[1][i] \in \text{InputList}[0])) \\
 & \wedge \forall i (\text{OutputList}[0][i] \leq \text{OutputList}[0][i + 1]) \\
 & \wedge \forall i (\text{InputList}[0][i] \leq \text{InputList}[0][i + 1]) \\
 & \wedge \forall i (\text{InputList}[1][i] \leq \text{InputList}[1][i + 1])
 \end{aligned}$$

The comparison data is shown in Table 5. 800 population size and 50 maximal generation are set in our approach. The experimental data shows that our approach performs better than neural network approach with small training data size and evenly with larger training data size. It also shows that the IOLRL can well-described the behaviors of the three benchmarks and the evolved relations can approximate the performance of the manually specified relations.

7 Conclusion

In this paper, we model PUTs in black-box manner and design IOLRL to formally describe the relations between the input and output list of PUTs. We use genetic programming to evolve relations in IOLRL that can well separate *pass* and *fail* test cases. With these relations and some labelled test cases, we build an SVM model as a test oracle. The advantage of our approach is only few test cases are needed to construct a test oracle. This is because we increase the variable dimensions of the test cases by supplying large set of GP population to evaluate the test cases. The potential problem is our designed IOLRL is not sufficient to effectively describe the I/O relations that can well separate the *pass* and *fail* test cases. However the experiments show the designed IOLRL is sufficient for the three benchmarks. The performance gap between the evolved relations and the manually specified relations can be compensated with better configurations of the GP procedure. In summary, this paper proposes a novel approach to evolve a test oracle with test cases and the experimental data shows our constructed test oracle performs well.

References

1. Brown, D.B., Roggio, R.F., Cross II, J.H., McCreary, C.L.: An automated oracle for software testing. *IEEE Transactions on Reliability* 41(2), 272–280 (1992)
2. Chang, C.-C., Lin, C.-J.: LIBSVM: a library for support vector machines (2001), Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>
3. Chen, T.Y., Ho, J.W.K., Liu, H., Xie, X.: An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics* 10 (2009)
4. Dustin, E., Rashka, J., Paul, J.: Automated software testing: introduction, management, and performance. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)

5. Edvardsson, J.: A survey on automatic test data generation. In: Proceedings of the Second Conference on Computer Science and Engineering in Linköping, October 1999. ECSEL, pp. 21–28 (1999)
6. Jones, E.L., Chatmon, C.L.: A perspective on teaching software testing. In: Proceedings of the seventh annual consortium for computing in small colleges central plains conference on The journal of computing in small colleges, USA, pp. 92–100. Consortium for Computing Sciences in Colleges (2001)
7. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
8. Peters, D.K., Parnas, D.L.: Using test oracles generated from program documentation. IEEE Transactions on Software Engineering 24(3), 161–173 (1998)
9. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming (2008), Published via, <http://lulu.com>, and freely available at <http://www.gp-field-guide.org.uk> (With contributions by J. R. Koza)
10. Vanmali, M., Last, M., Kandel, A.: Using a neural network in the software testing process. Int. J. Intell. Syst. 17(1), 45–62 (2002)
11. Vapnik, V.N.: Statistical Learning Theory. Wiley Interscience, Hoboken (1998)
12. Walker, M.: Introduction to genetic programming (2001)
13. Wang, Y.-C.F., Casasent, D.: A hierarchical classifier using new support vector machine. In: ICDAR, pp. 851–855. IEEE Computer Society Press, Los Alamitos (2005)
14. Weinbrenner, T.: Genetic programming techniques applied to measurement data. Diploma Thesis (February 1997)

Automated Driver Generation for Analysis of Web Applications

Oksana Tkachuk and Sreeranga Rajan

Fujitsu Laboratories of America,

Sunnyvale, CA USA

{oksana,sree.rajan}@us.fujitsu.com

Abstract. With web applications in high demand, one cannot underestimate the importance of their quality assurance process. Web applications are open event-driven systems that take sequences of user events and produce changes in the user interface or the underlying application. Web applications are difficult to test because the set of possible sequences of user inputs allowed by the interface of a web application can be very large. Software model checking techniques can be effective for validating such applications but they only work for closed systems. In this paper, we present an approach for closing web applications with a driver that contains two parts: (1) the application-specific Page Transition Graph (PTG), which encodes the application's possible pages, user and server events, their corresponding event-handlers, and user data and (2) the application-independent PTG-based driver, which generates test sequences that can be executed with analysis tools such as Java PathFinder (JPF). The first part can be automatically extracted from the implementation of a web application and the second part is written once and reused across multiple web applications belonging to the same framework. We implemented our approach in a driver generator that automatically extracts PTG models from implementation of JSP-based web applications, checks the extracted PTGs for navigation inconsistencies, and enables JPF analysis. We evaluated our approach on ten open-source and industrial web applications and present the detected errors.

1 Introduction

Web applications are open event-driven systems that take sequences of user events (e.g., button clicks in a browser window) and produce changes in the user interface (e.g., transition to a different page) or the underlying application (e.g., the shopping cart becomes empty). Web applications are difficult to test because the set of possible user event sequences allowed by the interface of a web application can be very large.

Current approaches to testing web applications often rely on constraining the analysis based on a formal model (e.g., UML in [13], WebML in [3], state machines in [1,7,9,14]). Some approaches rely on user specifications (e.g., [3,7]) or models extracted using run-time (e.g., [8,11,13]) or static analysis (e.g., [9,10]).

All approaches have their strengths and weaknesses. User specifications can capture the behavior that may be difficult to extract automatically but require manual work. Run-time crawling techniques are designed to visit web pages automatically. However, without user guidance (e.g., specification of user input data), crawlers may not be able to visit all possible pages of the application and the extracted models may miss some behavior. Static analysis techniques work well for specific domains and extracting specific features. For example, the approach in [10] extracts control flow graphs for web applications written in PLT Scheme and checks their navigation properties with a specialized model checker.

At Fujitsu, we have successfully used Java PathFinder (JPF) [2] to check business logic properties of web applications [12]. Our previous approach relied on user specifications describing (1) sequences of user events using regular expressions and (2) mappings from user events to the underlying application's event handlers that process them. Using these specifications, we automatically generated drivers that set up the event-handling code of the application under test, generated and ran test sequences with JPF. However, using this approach, the interface of the web application (e.g., its pages, buttons, and links) is abstracted away, since JPF cannot handle non-Java components and the generated drivers test the event-handling code of the application directly. In addition, the necessity to specify use case scenarios hindered the usability of the approach within the Fujitsu's testing teams.

To address the above limitations, we needed (1) to encode possible use case scenarios and event-handler registration information in a model, with an option to represent this model in Java, so JPF could handle it and (2) to increase usability by extracting this model automatically. In this paper, we present an approach to closing web applications with a driver that contains two parts: (1) the application-independent Page Transition Graph (PTG), which encodes the application's possible pages, user and server events, their corresponding event-handlers, and user data and (2) the application-independent PTG-based driver, which generates test sequences that can be executed with analysis tools such as JPF. By splitting the driver into two parts, we enable automation: (1) using framework-specific knowledge, the PTG can be automatically extracted from implementation of web applications and (2) the PTG-based driver is written once and reused across multiple web applications belonging to the same framework.

We implemented our driver generator and evaluated it on ten open-source and industrial JSP-based web applications. For each application, the driver generator (1) extracts and visualizes its PTG, (2) checks the extracted PTG for navigation errors and (3) generates and executes test sequences with JPF. To the best of our knowledge, this is the first driver generator that automatically extracts both navigation and event-handling aspects of the web applications and enables checking web applications with JPF.

This paper is organized as follows. The next section describes the PTG model and its usage for analysis of general web applications. Section 3 describes automated PTG extraction for JSP-based web applications, using Struts¹ as an

¹ <http://struts.apache.org/>

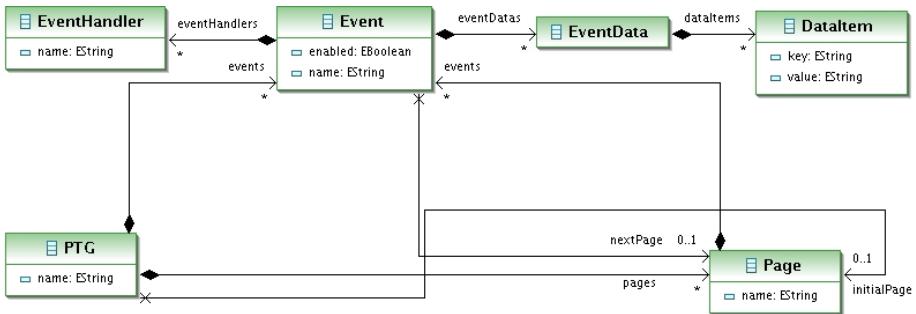


Fig. 1. PTG Class Diagram

example framework. Section 4 describes implementation of the PTG extractor. Section 5 presents experiments for ten open-source and industrial examples, including a discussion on the detected errors and limitations of the current approach. Section 6 describes the related work and Section 7 concludes.

2 Approach

In this section, we describe the PTG model, which is general enough to be applicable to various types of event-driven applications (e.g., GUIs). The goal of the PTG is to encode the navigation information and enable the registration of the application event handlers and generation of sequences of user events to drive the event-handling code of the web application under test. The PTG model is a graph with nodes corresponding to possible pages of the web application and transitions corresponding to possible user and server events, along with information about the event-handlers, event data, and possible pre-conditions for those events that require it. Equivalently, the PTG can be represented by a finite state machine.

Figure 1 shows a UML class diagram of the main elements of the PTG model designed based on our previous experience with model checking J2EE and GUI applications using JPF. A populated PTG encodes application-specific information about the application's set of available pages. Some pages can be marked as initial. Each **Page** contains a set of events attached to it. Each **Event** represents a possible transition to another page. In this paper, we concentrate on user events such as clicks on links and buttons and server events such as automatic redirection from one page to another. Some events may not be always enabled (e.g., buttons and links may become disabled depending on some condition). To reflect this possibility, **Event** declares a field `enabled`. Button click events are usually handled by the registered event handlers, therefore, events may have a set of **EventHandler**s associated with them. Some events may require user data, when filling out a form or writing into a text box. To reflect this possibility, an event may have a set of **EventData**, filled with **Dataitem**s, attached to it. The **EventData** is designed to hold keys and values, which correspond to the names

```

public class EnvDriver {
    public static HttpServletRequest req;
    public static HttpServletResponse res;
    public static AbstractPTG ptg;
    public static void main(String[] args){
        init(); //get populated ptg
        Page initPage=ptg.getInitPage();
        processPage(initPage);
    }
    public void processPage(Page p){
        List eList=p.getEvents();
        int nEvents=eList.size();
        //nondeterministic choice over events
        int eIndex=Verify.random(nEvents-1);
        Event e=eList.get(eIndex);
        if(e.isEnabled())
            processEvent(e);
    }
}

public void processEvent(Event e){
    List dList=e.getEventData();
    int nData=dList.size();
    //nondeterministic choice over data
    int dIndex=Verify.random(nData-1);
    EventData ed=dList.get(dIndex);
    req=new HttpServletRequestImpl();
    processEventData(ed, req);
    processEventHandling(e, req);
}

public Page processEventHandling(
    Event e, HttpServletRequest req){
    for(EventHandler h:e.getHandlers()){
        res=h.perform(req, res);
        nextPage=e.getNextPage(res);
        processPage(nextPage);
    }
}
}

```

Fig. 2. PTG-Based JPF Driver (excerpts)

of the text fields (e.g., `userid` and `password` on a login form and the user values entered in those fields).

2.1 PTG Analysis

The PTG model offers two levels of information: (1) navigation and (2) navigation with event-handling. The navigation level, containing information about possible page transitions, can be used to perform analysis with respect to navigation properties. This analysis does not require the business tier of the application under test to be part of the model, thus, it can be scalable and fast.

One can check various reachability properties (e.g., Page A is reachable from page B). However, such checks require selecting specific page names from a list of application pages. To simplify this step even further, we designed several checks that look for possible violations without user specifications. In Section 5, we present our experience with the following types of possible errors:

- Unreachable pages: pages unreachable from the initial page
- Ghost pages: referenced pages that do not exist
- Undefined transitions: transitions whose references do not match any of the available definitions (e.g., due to a typo)

2.2 PTG-Based Analysis

To check the business tier of the application under test, we developed a reusable application-independent driver that, given a populated application-specific PTG, traverses the PTG and generates and executes sequences of user events. Figure 2 shows a driver, tuned to J2EE event-handling APIs. The driver implements the following methods: `processPage()` calls `processEvent()` for each enabled event available on that page; `processEvent()` creates an `HttpServletRequest` object, populates it with event data, using `processEventData()`, and executes the event handling code; `processEventHandling()` calls the event handling method of the

event handler and returns a next page depending on the result. The algorithm recursively calls `processPage()` on the next page, thus traversing the PTG using Depth First Search (DFS). This algorithm can be augmented with conditions to vary the flavor of traversal (e.g., a bound can be used to limit the length of each generated sequence).

The implementation of the driver, along with the populated PTG, implementation of the web application under test, and database stubs (developed in our previous work [12]) are fed to JPF. One can use JPF’s listener framework to implement listeners that support property specification based on temporal logic (e.g. “the cart becomes empty after checkout”) or, in the absence of specifications, check for run-time errors or coverage. In section 5, we report the number of generated test sequences and coverage in terms of PTG nodes and transitions traversed by the driver.

One key feature of the driver is its ability to encode a *nondeterministic* choice over a set of events attached to a page or a set of data available for population of `HttpServletRequest`. To encode the nondeterministic choice, we use JPF’s modeling primitive `Verify.random(n)`, which forces JPF to systematically explore all possible values from a set of integers from 0 to n. Another feature of the driver is its extensible APIs. One can override certain methods to customize the driver according to the desired traversal algorithm (e.g., keeping track of visited transitions to avoid loops), the underlying analysis framework (e.g., to execute with the Java JVM by changing nondeterministic choices to for-loops), and the framework used to encode the event-handling APIs (e.g., Struts, as presented in the next section).

3 PTG Extraction

Driver generation in general is a hard problem. By concentrating on specific frameworks, we can achieve a high degree of automation. In this section, we present an automated PTG extraction technique for Java-based web applications that encode their page transitions in JSP and XML files. As an example of such a domain, we use Struts-based applications. The choice of the Struts framework was motivated by a request from Fujitsu’s development teams that use Struts to speed up the development time. In this section, we describe the Struts framework, give a small example and present the PTG extraction technique.

3.1 Struts Framework

Struts² is an open-source framework based on the Model-View-Controller (MVC) design pattern. The view portion is commonly represented by JSP files, which combine static information (e.g., HTML, XML) with dynamic information (e.g., Java as part of JSP scriptlet).

² There are two editions of Struts: Struts1 and Struts2. In this section, we describe Struts1; Struts2 is handled similarly.

The controller is represented by the Struts servlet controller, which intercepts incoming user requests and sends them to appropriate event-handlers, according to action mappings information specified in the XML descriptor file usually called `struts-config.xml`. In Struts, the request handling classes are subclassed from the `Action` class in the `org.apache.struts.action` package. Their event-handling method is called `execute()`. Actions encapsulate calls to business logic classes, interpret the outcome, and dispatch control to the appropriate view component to create the response. Form population is supported by the `ActionForm` class, which makes it easy to store and validate user data.

3.2 Example

To demonstrate our technique, we use a small registration example that allows users to register and login to their accounts, followed by a logout. This is a simplified version of the original application³. To simplify presentation, we removed several pages and events from the original application. In Section 5, we present experiments for both versions of this example.

The registration example encodes its page transitions using two XML files (`web.xml` and `struts-config.xml`) and six JSP pages: `index.jsp` (the initial page, marked in the `web.xml` configuration file), `welcome.jsp`, `userlogin.jsp`, `login-success.jsp`, `userRegister.jsp`, and `registerSuccess.jsp`. The event-handling part of the example contains four `Action` classes and two `ActionForms`.

The PTG extraction approach has two major steps: (1) parsing JSP/XML/-Java files, mining relevant information from them, and storing the information in the convenient form, commonly referred to as the Abstract Syntax Tree (AST) and (2) building PTG based on the previously mined information. Next, we describe these steps.

3.3 Extracting PTG-Related Data

The parsing step mines information from (1) JSP files, (2) XML configuration files, and (3) class files that encode `Actions` and `ActionForms`.

Analyzing JSP. The first step parses all JSP files of the application. Each JSP page corresponds to a `Page` node in the AST. Each JSP file is scanned for information about possible user and server events, encoded statically in JSP. Figure 3 (left) shows examples of such encodings on the pages of the registration example: `index.jsp` redirects to `welcome`, the `welcome.jsp` page contains links back to itself, to `userRegister.jsp`, and `userlogin.jsp`, the `userlogin.jsp` page contains a reference to the `/userlogin` action, and `loginsuccess.jsp` contains a redirect to another page.

To find references to possible user and server events, the parsers need to know the types of encoding to track. Figure 4 shows examples of such encodings: link and form JSP/HTML tags and attributes, redirect tags and scriptlet keywords to

³ <http://www.roseindia.net/struts/hibernate-spring/project.zip>

```
//from index.jsp                                //from struts-config.xml
<logic:redirect forward=welcome               <action mappings>
//from welcome.jsp                            <action path="/Welcome"
<html:link page="/Welcome.do">                forward="/pages/Welcome.jsp"/>
    <font color="white">Home</font></html:link>   </action>
<html:link page="/pages/user/userRegister.jsp"> <action path="/userlogin"
    <font color="white">Register</font></html:link> name="UserLoginForm"
<html:link page="/pages/user/userlogin.jsp">    type="UserLoginAction">
    <font color="white">Login</font></html:link>   <forward name="success"
//from userlogin.jsp                           path="loginsuccess.jsp"/>
<html:form action="/userlogin" method="post">  <forward name="failure"
//from loginsuccess.jsp                         path="userlogin.jsp"/>
if(userid==null)                            </action>
    response.sendRedirect("../userlogin.jsp")  </action-mappings>
```

Fig. 3. Example JSP tags and XML definitions (excerpts)

```
//links
LINK_TAG = <html:link;<c:url;<s:url
LINK_TAG_ATTRIB = href
//forms (actions)
FORM_TAG = <html:form;<s:form
//redirects
REDIRECT_TAG = <logic:redirect;<jsp:forward
REDIRECT_SCRIPTLET = response.sendRedirect
//includes
INCLUDE_TAG = <jsp:include;<jsp:param
INCLUDE_SCRIPTLET = include
```

Fig. 4. JSP Parser Configuration File

track redirect and inclusion relationships (i.e., one JSP page can include another and display forms and links available on the included page). These encodings allow parsers to find references to possible user and server events and store them as part of the **AST Events**. Each **AST Event** stores information about its **path**, which can be a reference to the next JSP page or an action, defined in the XML configuration files. When the **path** refers to a URL or a file not related to PTG (e.g., an image), the parsers filter out such events based on the naming conventions.

Analyzing XML. XML configuration files contain various definitions needed at deployment time. For example, `web.xml` contains information about the naming conventions and initial pages, whereas `struts-config.xml` contains action definitions. As an example, we describe the definition of the `/userlogin` action, shown in Figure 3 (right), referenced on the `userlogin.jsp` page: (1) this is the form submission event, taking `UserLoginForm`, (2) it is handled by the event-handler of type `UserLoginAction`, (3) the outcome of this event depends on whether the event handling code returns "success" or "failure". In case of "success", the next page is `loginsuccess.jsp`; in case of "failure", the next page is `userlogin.jsp`. Note that in this particular case, the **path** of each outcome references a JSP page. In general, it may reference another definition, available in the given or another XML configuration file.

The XML parsers parse and store all of the XML information as part of the **AST Definitions**. Definitions describing form submission events require additional data, used to populate `ActionForm` objects. This information can be mined from several sources, including `ActionForm` classes themselves.

Analyzing Java Classes. This step finds and loads all `ActionForm` classes of the application. For each application form, e.g., `UserLoginForm`, it loads its class file

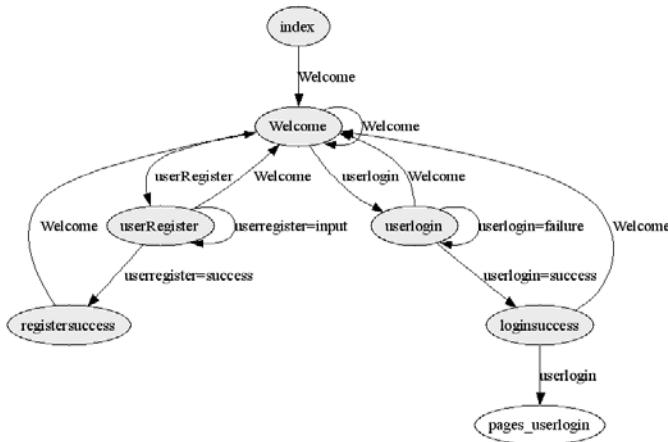


Fig. 5. Example PTG (Visualization)

and, using reflection APIs, finds all of its fields, e.g., `userid` and `password`. Using naming conventions, each field can be set through a field-specific setter method. For example, `setUserId(String)` sets the `userid` field and `setPassword(String)` sets the `password` field. Thus, after discovering field names, we are able to generate code that sets these fields to values that represent user values entered into the corresponding text fields. To generate user values, this step relies on values specified in a file (e.g., based on symbolic execution that supports strings [4]). In the absence of specifications, this step generates default values (e.g., common corner cases such as empty, non-empty strings).

3.4 PTG Construction

After the parsing step, the PTG construction step takes the AST, containing the information about all JSP pages, their possible user and server events, and available XML definitions, and populates the PTG. For each AST Page, the algorithm creates a PTG Page. Then, for each reference to the user or server event available on each AST page, the algorithm tries to resolve the reference, based on the encoded path of the event. If the path references a JSP page, then a lookup mechanism is invoked to find that page. If the page is not found, a new ghost page is created and set as a destination page of the event. If the path does not reference a JSP page, then it is looked up under the definitions. If the definition is not found, the next page of the event is set to the special `Undefined` ghost page, otherwise, the definition is used to resolve the possible destination pages of the event. Each definition updates the event according to its domain-specific encoding, e.g., the `Action` definitions from `struts-config.xml` generate `EventHandlers` according to the `type` information and calculate destination pages according to the information encoded in `forwards`. This part of the algorithm is recursive, since a definition may reference other definitions. After the event construction is finished, the event is added to its source page.

By construction, our approach extracts an overapproximation of the page transitions, with respect to static JSP and XML encodings specified to the tool. Currently, the tool treats all transitions as *possible* and does not keep track of conditions under which some transitions may become enabled or disabled. We note that the model of `Event` includes the `enabled` field and `isEnabled()` method, used by the PTG-based driver. It is currently the limitation of the JSP parser, which does not extract conditions that may affect enabledness of some events at run-time.

Even with overapproximations, the extracted PTG is useful for visualization and detecting potential errors. Figure 5 shows visualization of the page transition portion of the PTG extracted for the registration example, which shows one (white) ghost page. It is calculated when processing `response.sendRedirect()` event on the `loginsuccess.jsp` page. This event refers to the ".../userlogin.jsp" page, which does not exist; the correct path is `userlogin.jsp`. Evaluating this potential error, we find that this transition has a condition associated with it: `if(userid==null)`. Under normal usage, this error is not exercised, however, it is still possible to see the error at run-time. When accessing the `loginsuccess.jsp` page directly, the server throws an exception with "the requested resource not available" message. Such errors often appear after refactoring or due to typos and are difficult to find using traditional testing techniques.

Note that, the PTG analysis with respect to navigation properties requires no extensions for Struts domain. To perform the PTG-based analysis of Struts-based applications, the generic driver, presented in section 2.2, needs to be extended to handle domain-specific event-handling APIs, for example the method `processEventHandling()` needs to be overridden:

```
public Screen processEventHandling(Event event, HttpServletRequest req){
    List handlers = event.getEventHandlers();
    for(Action action : handlers){
        forward = action.execute(...,form, req, res);
        nextPage = getNextPage(event, forward); ...
    }
}
```

4 Implementation

We implemented the PTG extractor with the following modules:

Parsers: The JSP parser is automatically generated by JavaCC, based on the grammar capable of parsing JSP/HTML/XML tags, scriptlet blocks, and various types of comments. The JSP parser extracts information based on the tags specified in a separate file (similar to one in Figure 4). One can add additional tags, based on the domain-specific encodings. XML parsers are based on the Digester framework⁴. We currently support parsing of `web.xml`, and configuration files used by Struts, Struts2, and Tiles⁵ frameworks.

Generators: Given the AST, generators populate the PTG data structure. We implemented the PTG data structure using the Eclipse Modeling Framework

⁴ <http://commons.apache.org/digester/>

⁵ <http://tiles.apache.org/framework/index.html>

(EMF)⁶. EMF supports model specification using Java interfaces. Given Java interfaces, EMF automatically generates their implementation and provides additional facilities, such as writing the model in XML and loading it from XML to Java. The PTG APIs can be used as a library.

Printers: Given the populated PTG, printers generate its various representations. Currently, we support PTG generation in Java, used by the PTG-based JPF driver, XML, used to populate page and event names for property specifications, and the Dot⁷ representation, used for visualization.

Checkers: We implemented various checkers, which take a populated PTG and collect features that may signal possible errors (e.g., unreachable and ghost pages and undefined transitions). The checkers print their output in XML.

The PTG generator has an extensible architecture: one can plug in their own implementation of parsers and code generators. The entire codebase is around 10K LOC, of which 4K LOC belongs to the auto-generated code (i.e., the EMF-generated PTG code and the JavaCC parser). The PTG generator is fully automated: one only needs to specify the application directory of the example under test.

5 Experience

We evaluated our driver generator on a collection of ten examples. Before setting up the experiments, we had the following research questions:

PTG Quality: What is the quality of the PTG generator? Does it miss any transitions? Does it generate spurious transitions? Does it enable error detection? What is the level of coverage achieved by the PTG-based driver at run-time?

Human/Tool Cost: What is the level of automation? What is the level of manual work?

5.1 Case Studies

To perform evaluation, we downloaded several open-source Struts-based examples: the simplified and the original registration example, described in section 3.2, *cookbook*, *artimus* and *polls* from sourceforge⁸, *petstore*⁹, *personalblog*¹⁰, and *cart*¹¹. In addition, we were given two Fujitsu’s internal applications: a sample Project Management application, called in this paper *FujitsuPM*, and a sample Document Management application, called *FujitsuDM*. All examples can be deployed using the Tomcat server.

⁶ <http://www.eclipse.org/emf>

⁷ <http://www.graphviz.org/>

⁸ <http://sourceforge.net/projects/struts/files>

⁹ <http://www.jwebhosting.net/servlets/jpetstore5/index.html>

¹⁰ <http://suif.stanford.edu/~livshits/work/securibench/download.html>

¹¹ <http://www.roseindia.net/shoppingcart/cart1.1.zip>

Table 1. Examples Static Data

ID	AppName	Frameworks	LOC	C1 (All/Act/Form)	JSP/XML
1	<i>simregister</i>	Struts	615	15/4/2	6/2
2	<i>register</i>	Struts	709	34/5/3	12/2
3	<i>cookbook</i>	Struts, Tiles	681	18/1/0	17/4
4	<i>artimus</i>	Struts, Tiles	799	16/1/1	20/3
5	<i>petstore</i>	Struts	1,705	26/5/5	21/2
6	<i>personalblog</i>	Struts, Tiles	807	24/11/5	23/3
7	<i>polls</i>	Struts, Tiles	8,980	48/13/0	35/3
8	<i>cart</i>	Struts, Tiles	3,873	57/16/11	49/3
9	<i>FujitsuPM</i>	Struts2	9,876	67/9/0	25/3
10	<i>FujitsuDM</i>	Struts, Tiles	18,277	129/17/9	29/3

Table 1 shows static information about the case studies. The **Frameworks** column describes the types of frameworks used by the front end of each application. The **LOC** column gives the number of lines of code; the **C1** column gives the number of all classes, followed by the number of **Action** classes, followed by the number of **ActionForm** classes. The **JSP/XML** column gives the number of application’s JSP pages and XML configuration files relative to the PTG population.

5.2 Experiment

For each case study, we performed the following steps:

1. PTG Extraction: we ran the PTG extractor with the default values used for population of the forms. This step generates PTG and prints it using the following representations: Java (used by JPF), XML, and Dot.
2. PTG Analysis: we enabled PTG checkers looking for various possible violations and features. We present our finding for the following features described in section 2.1: unreachable pages, ghost pages and undefined transitions. Checking these features requires no user specifications.
3. PTG-based Analysis: We ran the driver using the traversal algorithm that avoids loops by keeping track of visited transitions. We used the Java representation of PTG, generated in the first step. We measured page and transition coverage achieved by the driver.

All experiments were run on a Linux machine, with 2G RAM and 2.66 GHz processor. Table 2 shows the collected data. The **PTG** column shows the size of the extracted PTG in terms of pages/nodes and events/transitions. The **Time** column shows the time, in min:sec format, the PTG extractor takes to parse JSP/XML/Java files and build a PTG for each example. The **Errors** column presents possible errors found in each PTG: pages unreachable from the initial page, followed by ghost pages, followed by the number of undefined transitions. The next columns present data for generation of test sequences based on running the PTG-based driver with JPF. The **Sts/Mem/Time** column gives the number of end states as reported by JPF, which corresponds to the number of

Table 2. Examples Experiment Data

ID	AppName	PTG (n/tr)	Time	Errors	Sts/Mem/Time	Cov(n/tr)
1	<i>simregister</i>	7/13	00:02	0/1/0	48/54/00:03	7/13
2	<i>register</i>	13/25	00:03	2/1/0	192/75/00:04	11/25
3	<i>cookbook</i>	17/18	00:03	5/0/0	15/56/00:02	12/17
4	<i>artimus</i>	20/29	00:03	9/0/0	256/70/00:11	10/21
5	<i>petstore</i>	22/87	00:04	2/1/8	7,671/106/01:30	20/85
6	<i>personalblog</i>	24/83	00:03	3/1/3	11,618/122/01:17	21/61
7	<i>polls</i>	36/113	00:07	5/1/1	10,081/106/02:04	28/76
8	<i>cart</i>	55/165	00:08	31/6/3	6,345/105/01:11	24/75
9	<i>FujitsuPM</i>	27/71	00:05	11/3/11	1,945/76/00:26	16/57
10	<i>FujitsuDM</i>	30/97	00:07	7/1/3	7,452/107/01:03	22/73

explored paths, and JPF resources in terms of the memory used (in MB) and time. The last, **Cov**, column gives the run-time coverage over the PTG nodes and transitions as explored by JPF.

The data show that many PTG models contain unreachable pages, ghost pages, and undefined transitions. For example, the *FujitsuPM* PTG contains 11 unreachable pages, all going to the special `Undefined` ghost page, two other ghost pages and 11 undefined transitions. We reported these results to the development team of the application. The team was surprised but confirmed that all errors were real and, possibly, appeared in the sample application after the code refactoring. The team members requested a tool demo and were pleased with the speed of the PTG extractor and the ease of its use. Currently, the tool is being evaluated for possible integration into their testing environment.

The last four columns of Table 2 show the data for the run-time execution of the PTG-based driver, as executed by JPF. The last column shows coverage in terms of the driver's ability to visit pages and execute events. The numbers show that the driver misses some pages and transitions. Manual evaluation shows that most of the missed pages are due to their unreachability from the initial page. The *cart* example shows particularly large amount of unreachable pages. Inspecting the example, we found that the graph contained 2 components, possibly corresponding to the user and the administrator modules. The administrator pages were not reachable from the default initial page of the application. Such issues can be corrected by specifying additional initial pages to the PTG generator.

The driver also misses some transitions due to the limited data values, used to populate `ActionForms`. Since the data population is done fully automatically, using default corner-case values, it is possible for the driver to miss some outcomes of the `execute()` event-handling code. Despite such limitation, the end states number reported by JPF shows that the PTG-based driver is capable of executing hundreds of test sequences within seconds.

5.3 Discussion

In this section, we address our research questions.

PTG Quality: We manually evaluated the extracted PTGs and found no missing transitions with respect to the encodings specified to the tool. We also manually evaluated all reported errors and found no false warnings, except for the `Undefined` ghost page, which is used by the model to visualize the destination page of the undefined transitions. Most reported ghost pages show up on rare executions outside of normal use case scenarios. However, these errors are reproducible at run-time, when accessing some pages directly, without going to the initial page first (as explained in section 3.4, using the *simregister* example).

Even with possible overapproximations, we believe, the PTG model is useful. It enables automatic checking of possible navigation errors and enables validation of the application under test using the PTG-based driver. While the PTG model may encode more executions than possible at run-time, it can serve as a starting navigation and event-handling model of the application PTG, which can be manually edited with conditions to restrict some of the overapproximations.

Human/Tool Cost: Our experiment consists of three fully automated steps. Steps 2 and 3 can be enhanced with user-supplied properties. However, even without user specifications, our technique is capable of finding suspicious features such as unreachable and ghost pages or undefined transitions. We found several such violations in many case studies. These violations would have been hard to find with traditional testing approaches (e.g., testing or run-time crawling would not uncover unreachable pages).

Threats to Validity: The set of case studies is limited yet representative. Case studies were picked by people not involved with the development of the PTG generation approach. The examples were picked based on their accessibility and the frameworks used for their implementation.

6 Related Work

Many approaches to testing web applications require users to specify requirements (e.g., [7,10]). User specifications can capture behavior based on high-level requirements, not system implementation, which may be faulty. However, they require manual work, which could also be prone to errors. The PTG model extracted in our approach can serve as a starting point for specifications in such approaches.

Run-time analysis can be used to extract the model, while executing the application under test. For example, Memon et al. [11] extract an Event Flow Graph (EFG) while executing a GUI application. The extracted EFG is used to generate sequences of events and feed them back to the GUI application under test. Haydar [8] presents an approach to extract a finite automata model from execution traces. Without user guidance (e.g., specification of user input data), run-time approaches may not be able to visit all possible pages of the application and the extracted models may miss some behavior. To avoid such issues, some approaches require user guidance (e.g., [1,13]).

Static analysis techniques work well for specific domains and extracting specific features. For example, Kubo et al. [9] present an automated technique for

extracting page transitions from Struts applications. Unlike in our work, this approach extracts and model checks, with SPIN, page transitions only, whereas in our work, we also extract the information about the event-handling classes and event data, which can be used to drive the underlying event-handling code of the application under test with analysis tools such as JPF. Yuen et al. present web automata [14], a behavioral model for MVC-based applications, similar to PTG. They propose to extend `struts-config.xml` into another XML file that represents an automaton, which can be used to check reachability requirements. However, they do not present automation for their approach.

Recent work on interface discovery for web applications [5,6] uses static analysis [6] and symbolic execution [5] to analyze Java servlets to calculate possible data inputs, represented by sets of input parameters (e.g., `username` and `password`) and their possible values. This analysis is designed specifically for servlets, which usually check user input. In contrast, our analysis focuses on generating sequences of user events allowed by the application’s interface. In spite of differences, their approach is complementary and can be used to enhance the PTG `EventData` with interesting user values.

7 Conclusions and Future Work

In this paper, we presented a driver generation approach for analysis of JSP-based web applications that (1) using the application implementation, automatically extracts a Page Transition Graph (PTG) that encodes page transitions, possible user and server events, their corresponding event-handlers and user data (2) visualizes and checks the PTG for presence of ghost pages, undefined transitions, and unreachable pages and (3) generates and executes test sequences with JPF. We evaluated the tool on ten Java applications, including two large industrial applications. We uncovered multiple navigation errors in many case studies, including sample applications from the Fujitsu development teams.

The landscape of frameworks used to develop applications is always changing. In this paper, we used the Struts framework as an example for domain-specific automation support, however, our tools can be extended to handle other frameworks and approaches. We are interested in (1) extending the JSP parsers with parsing of event conditions to prune possible spurious transitions, (2) evaluation of symbolic execution [4] as a technique to enhance generation of event data, (3) extending the driver to run with HttpUnit¹², and (4) combining our static approach for PTG extraction with a dynamic approach (e.g., crawling).

Acknowledgments

We are deeply grateful to the JPF team, especially Willem Visser and Peter Mehltz, for their constant support with JPF.

¹² <http://httpunit.sourceforge.net/>

References

1. Andrews, A.A., Offutt, J., Alexander, R.T.: Testing web applications by modeling with fsm. *Software and Systems Modeling* 4, 326–345 (2005)
2. Brat, G., Havelund, K., Park, S., Visser, W.: Java PathFinder – a second generation of a Java model-checker. In: Proceedings of the Workshop on Advances in Verification (July 2000)
3. Deutsch, A., Sui, L., Vianu, V., Zhou, D.: A system for specification and verification of interactive, data-driven web applications. In: SIGMOD 2006: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 772–774. ACM, New York (2006)
4. Ghosh, I., Rajan, S., Shannon, D., Khurshid, S.: Efficient symbolic execution of strings for validating web applications. In: Proceedings of the International Workshop on Defects in Large Software Systems (July 2009)
5. Halfond, W.G., Anand, S., Orso, A.: Precise interface identification to improve testing and analysis of web applications. In: ISSTA 2009: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, pp. 285–296. ACM, New York (2009)
6. Halfond, W.G.J., Orso, A.: Improving test case generation for web applications using automated interface discovery. In: ESEC-FSE 2007: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pp. 145–154. ACM, New York (2007)
7. Hallé, S., Ettema, T., Bunch, C., Bultan, T.: Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In: ASE, pp. 235–244 (2010)
8. Haydar, M.: Formal framework for automated analysis and verification of web-based applications. In: ASE 2004: Proceedings of the 19th IEEE International Conference on Automated software Engineering, pp. 410–413. IEEE Computer Society, Washington, DC, USA (2004)
9. Kubo, A., Washizaki, H., Fukazawa, Y.: Automatic extraction and verification of page transitions in a web application. In: APSEC 2007: Proceedings of the 14th Asia-Pacific Software Engineering Conference, pp. 350–357. IEEE Computer Society, Washington, DC, USA (2007)
10. Licata, D.R., Krishnamurthi, S.: Verifying interactive web programs. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, pp. 164–173. IEEE Computer Society, Washington, DC, USA (2004)
11. Memon, A., Banerjee, I., Nagarajan, A.: Gui ripping: Reverse engineering of graphical user interfaces for testing. In: WCRE 2003: Proceedings of the 10th Working Conference on Reverse Engineering, p. 260. IEEE Computer Society, Washington, DC, USA (2003)
12. Rajan, S.P., Tkachuk, O., Prasad, M.R., Ghosh, I., Goel, N., Uehara, T.: Weave: Web applications validation environment. In: ICSE Companion. Software Engineering in Practice, vol. 2, pp. 101–111 (2009)
13. Ricca, F., Tonella, P.: Analysis and testing of web applications. In: ICSE 2001: Proceedings of the 23rd International Conference on Software Engineering, pp. 25–34. IEEE Computer Society Press, Washington, DC, USA (2001)
14. Yuen, S., Kato, K., Kato, D., Agusa, K.: Web automata: A behavioral model of web applications based on the mvc model. *Information and Media Technologies* 1(1), 66–79 (2006)

On Model-Based Regression Testing of Web-Services Using Dependency Analysis of Visual Contracts

Tamim Ahmed Khan and Reiko Heckel

Department of Computer Sciences, Leicester University, UK
re o s e

Abstract. Regression testing verifies if systems under evolution retain their existing functionality. Based on large test sets accumulated over time, this is a costly process, especially if testing is manual or the system to be tested is remote or only available for testing during a limited period. Often, changes made to a system are local, arising from fixing bugs or specific additions or changes to the functionality. Rerunning the entire test set in such cases is wasteful. Instead, we would like to be able to identify the parts of the system that were affected by the changes and select only those test cases for rerun which test functionality that could have been affected.

This paper proposes a model-based approach to this problem, where service interfaces are described by visual contracts, i.e., pre and post conditions expressed as graph transformation rules. The analysis of conflicts and dependencies between these rules allows us to assess the impact of a change of the signature, contract, or implementation of an operation on other operations, and thus to decide which of the test cases is required for re-execution. Apart from discussing the conceptual foundations and justifications of the approach, we illustrate and evaluate it on a case study of a bug tracking service in several versions.

1 Introduction

Service-oriented systems pose new challenges to client-side testing [1]. Problems arise from the lack of access to and control over the implementation (let alone the code), which prohibit the use of white box techniques and may limit (due to the cost for using a service or the limited time available) the number of tests that can be executed [2].

Evolution in software systems is inevitable to keep them abreast with the changing needs of businesses. To assess and assure that there is no deviation of the existing functionality, regression testing uses a comprehensive set of test cases to reevaluate every new version. Such regression test suites are accumulated over time and can be large and costly to run [3]. In many cases, however, the impact of a particular evolution step is limited to a small part of the system, especially if maintenance is concerned with minor corrections or additions. In such cases it would be beneficial to select only those test cases for rerun which exercise parts of the system directly or indirectly affected by the changes.

Following the classification in [4], a test case in a regression test suite can be *obsolete* (*OB*) if it is no longer applicable to the new version, *Reusable* (*RU*) if it is still applicable and *required* (*RQ*) if it tests functionality affected by the changes. Given

two consecutive versions of the system $V1$ and $V2$ together with information about the changes from one to the other, the problem is therefore to classify a set of test cases executable over $V1$ into the three categories in such a way that any faults detected in $V2$ by executing RU are also detected running RQ only. In this paper we will provide a classification and argue for its correctness in the above sense both conceptually (based on a formalisation of the problem in terms of graph transformation systems) and by an evaluation through a small but non-trivial case study of a service in three versions.

Since code is not available, our treatment of the problem is based on model-based service descriptions at the level of interfaces. In this way we also abstract from details of the programming language, supporting the platform-independent nature of services. Semantic information at the interface level is expressed by means of typed graph transformation systems [5] presented as visual contracts [6]. This has the advantage of using a visual specification in line with mainstream software modelling languages such as the UML, while at the same time retaining a formal semantics and mathematical theory. Based in particular on the theory of conflicts, causality and concurrency of graph transformations we analyse data dependencies between and within test cases based on which we determine the impact of change and thus the set of required test cases.

We perform an evaluation of our method based on three versions of a Bug Tracking service adapted from an open source application in C#. Defining and classifying test cases for the three versions, we are interested in both the number of test cases saved by the classification and its continued ability to find all the faults. We use error seeding techniques to assess the quality of both the complete and reduced test sets.

The paper is organised as follows. Section 2 introduces the basic concepts of our approach, including the specification of visual contracts by graph transformation rules and the use of trace theory to provide an observational semantics appropriate for testing. Section 3 presents our evolution scenario and describes and applies our methodology. The evaluation is reported on in Section 4 while related work is discussed in Section 5 before we conclude the paper.

2 Visual Contracts and Trace Semantics

In model-based testing of services we are potentially concerned with three layers of abstraction: those of *implementation*, *interface model*, and *observable behaviour*. While the implementation is hidden, we will require information about changes, such as for which operations from the interface the implementation has been modified. The interface model details operation signatures and accompanying data types as well as describing the semantics of operations in terms of pre- and post conditions. Such descriptions may be available in diagrammatic form at design time, but also in machine-readable form at run time. The observable behaviour is expressed in terms of sequences of messages representing invocations to service operations, e.g., as part of a test case being executed. In order to define precise criteria for distinguishing different categories of test cases, we have to study the relation between interface models and observations.

Visual Contracts. We represent service interface models by typed graph transformation systems (TGTS). A TGTS $(TG \ Sig \ R)$ consists of a type graph TG modelling

the public data types available at the interface, a set of rule signatures Sig providing operation names with parameter declarations $p(x_1 : s_1 \dots x_n : s_n)$. Here $x_i : s_i$ represents a formal parameter x_i of type s_i . A set of rules R is associated to these signatures, describing the behaviour of the corresponding operations as visual contracts [7,8], i.e., pre- and post conditions $L \rightarrow R$ shown as object diagrams. We write $p(x_1 : s_1 \dots x_n : s_n) : L \rightarrow R$ if there is a rule $L \rightarrow R$ associated with an operation signature $p(x_1 : s_1 \dots x_n : s_n)$.

Example 1 (Bug Tracker service). In order to illustrate the use of these models we present a case study of a Bug Tracker service, to be used as a running example throughout this paper. Three consecutive versions of the service have been derived from an open source desktop application¹ by replacing its GUI by a service interface. Such a service could be useful, for example, in order to allow automatic bug reports through applications detecting faults or in order to integrate bug tracking data into higher level functions.

In its basic version, bug tracking serves the communication between developers, users, testing team, etc. Once a bug has been added by the user who discovered it, its status can be updated by the developers and testers until the issue is resolved. In addition to the interface provided to the user, we have also created an administrative interface to add, update, or delete projects and users. Both interfaces are listed in Fig. 1.

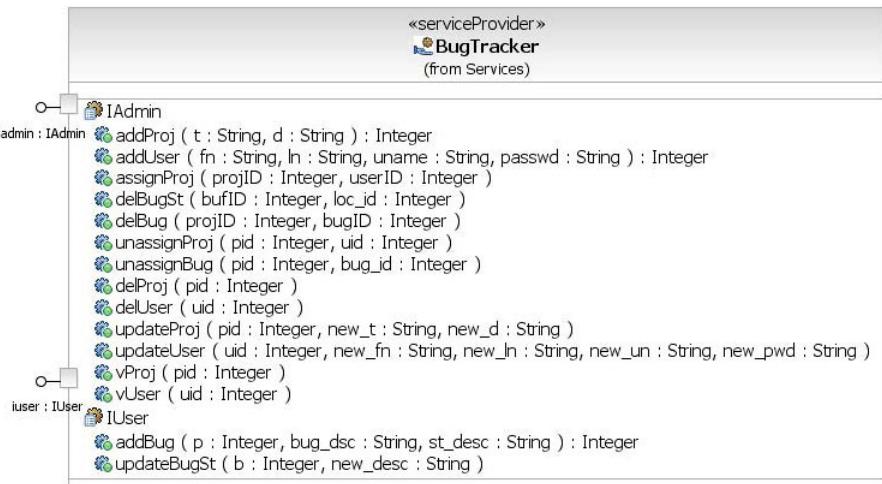


Fig. 1. Bug Tracking interfaces for Admin and User

A conceptual data model for the service, limited to the data visible to its clients, is presented in Fig. 2(a). Beside a top level class **BugTracker**, we find **User** and **Project** data as well as **Bug** and **Status** information. A selection of rules representing visual contracts are shown in Fig. 2(b). For example the **addBug** rule describes how a bug

¹ Available at <http://www.sss.org/e>

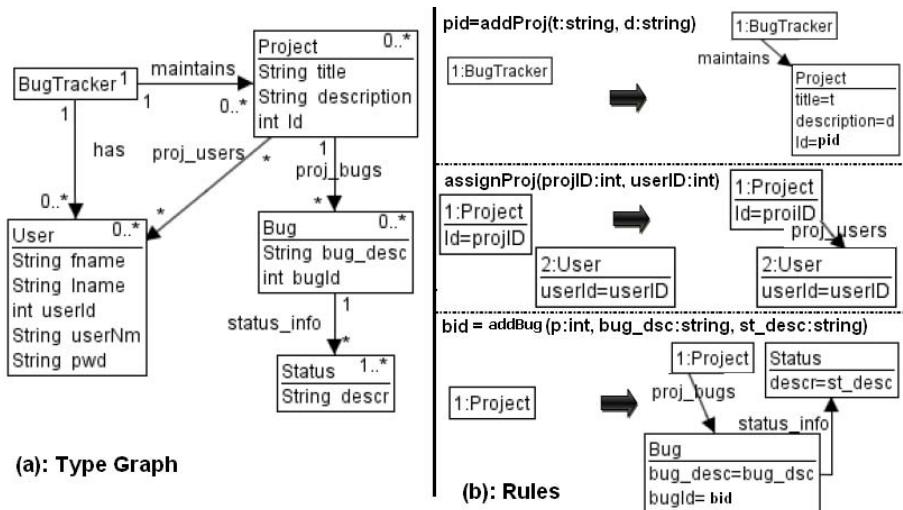


Fig. 2. Type graph and rules

report is added to the database, assuming an existing project and adding Bug and Status objects.

Observational Semantics. Graph transformation systems come with an execution semantics, i.e., we can simulate the implementation at the interface level by means of rule applications. Given a graph G representing a state of the system and an operation $p(x_1 : s_1 \dots x_n : s_n) : L \rightarrow R$, we can attempt an invocation $p(a_1 \dots a_n)$, substituting formal parameters x_i in $p(x_1 : s_1 \dots x_n : s_n)$ by actual data values a_i found in G_0 . If there exists a match $m : L \rightarrow G$ embedding L into G such that $m(x_i) = a_i$, i.e., m is compatible with the instantiation of parameters, the rule can be applied resulting in a *transformation step* $G \xrightarrow{p^m} H$. The observation or *label* of this step, $(G \xrightarrow{p^m} H) \vdash p(a_1 \dots a_n)$ is given by the rule name with actual parameters, while the state and the actual rule remain hidden. The set of all possible observations for the (implicit) signature Sig is denoted by O whereas the set of all possible sequences provided by Kleene closure of O is denoted by \overline{O} .

Assuming a start state represented by graph G_0 , selecting rules and matches we can produce a *transformation sequence* $G_0 \xrightarrow{p_1 m_1} G_1 \xrightarrow{p_2 m_2} \dots \xrightarrow{p_n m_n} G_n$. The set of all these sequences is $er(\cdot)$ and the observation function $\overline{\cdot}$ extends to such sequences, i.e., $\overline{\cdot} : er(\cdot) \rightarrow \overline{O}$. That means, (\cdot) is the sequence of labels obtained as observations of the steps of \cdot .

Example 2 (transformation sequence and observation). Consider the bug tracking system whose interface is shown in Fig. 1 and the type graph and rules are shown in Fig. 2 with a start state having only one project and one user as represented by graph G_0 in Fig. 3. Transformation sequence $G_0 \xrightarrow{addProj m_1} G_1 \xrightarrow{addUser m_2} G_2 \xrightarrow{assignProj m_3} G_3$ shown in Fig. 3 creates a new project and user and assigns the project to the user.

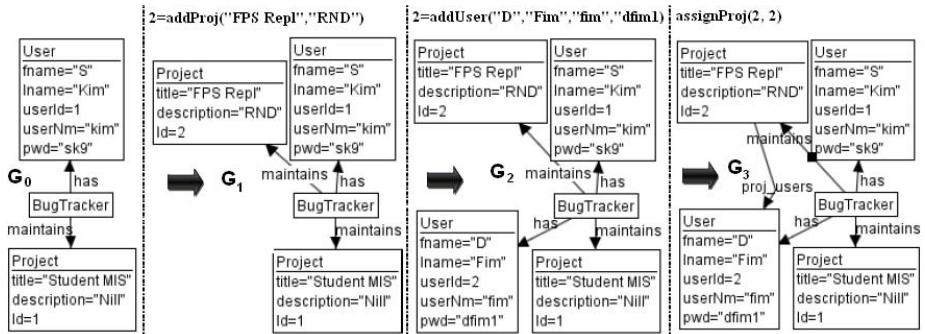


Fig. 3. Transformation sequence

The corresponding sequence of observations is $2 = \text{addProj}("FPS Repl", "RND")$; $2 = \text{addUser}("D", "Fim", "fim", "dfim1")$; $\text{assignProj}(2, 2)$ where return values are 2 in both $\text{addProj}(\quad)$ and $\text{addUser}(\quad)$.

In order to ensure that labels carry enough information for observations to reflect faithfully the behaviour at the interface level, we have to make a number of assumptions. First, we assume that all objects can be uniquely identified by their collection of attributes, and that these attributes are always fully defined in the states of the system. This is of course a requirement for the initial state, but also for the rules specifying the operations, which have to preserve these properties. Second, operation signatures need to carry enough parameters to identify uniquely the match of a transformation. This is satisfied, for example, if each signature lists *id* attributes for all elements of its rule's left-and right-hand side as parameters, thus specifying completely the embedding of the rule into graphs G and H . In most practical cases, however, parameters will only need to identify some anchor elements, which will then determine the other elements in the match and co-match. For example, as stated in the cardinality of 1 on the *status_info* association, a Status object will always refer to a unique Bug, so identifying the Status we implicitly know the Bug as well. These conditions are formalised in [9] in terms of morphisms of attributed graphs. If they are satisfied, the observation function π is called *faithful*.

Conflicts and Dependencies. In order to understand if two observations can be part of the same invocation sequence, or if they can occur in that sequence in a given order, we have to analyse causal dependencies and conflicts between transformations and represent them at the level of labels. At the model level, we say that

- a competing transformation $G^{p_2 m_2} H_2$ disables $G^{p_1 m_1} H_1$ if the match for p_1 is destroyed by the application of p_2 ;
- a consecutive transformation $G_1^{p_2 m_2} G_2$ requires $G_0^{p_1 m_1} G_1$ if the application of p_1 creates elements required for the application of p_2 .

These relations are essentially those of asymmetric event structures [10]. The asymmetry arises from the interplay of deletion and preservation, which is typical to all computational models distinguishing read and write access to resources.

If two steps are not in conflict or dependent, they are independent. Two independent consecutive steps can be swapped. The standard model of concurrency for graph transformation systems, based on the so called *shift-equivalence* $sh \subseteq Der(\) \cap Der(\)$, abstracts from the order in which independent steps are applied, considering all derivations equivalent that represent serialisations of the same concurrent process. The quotient $Der(\) / sh$ defines the set of concurrent derivations in the system.

In order to derive a concurrent *observational* semantics, following [9] we lift the *disables* and *requires* relations to the level of labels. For two labels l_1 l_2 and transformations $_1$ and $_2$ such that $l_i \in (_i)$, we write

- $l_1 \rightarrow l_2$ if $_2$ disables $_1$;
- $l_1 \rightarrow l_2$ if $_2$ requires $_1$;

If l_1 l_2 are unrelated by \rightarrow and \rightarrow , they are independent, written $l_1 \perp l_2$.

In order to calculate conflicts and dependencies at the level of labels we make use of the critical-pair analysis technique using AGG [11]. Critical-pair analysis provides us with the minimal set of graphs, such that all possible overlapping situations between the left- and the right-hand sides of the rules are recorded. It captures *potential* conflicts and dependencies between rules, rather than (labels representing) transformation steps. Therefore, the result is an over-approximation of the actual dependencies at the level of the labels, specifically where more complex conditions on data values are used. Since we are working at the level of signatures, we represent the overlapping of rules as a relation between the parameters identifying those overlapping graph elements.

Example 3 (conflicts and dependencies on labels). For a small set of labels we have illustrated these relations in Table 1. An entry in a row labelled by l_1 and a column labelled by l_2 represents the relation between l_1 and l_2 . Each cell can contain either or both of \rightarrow or \rightarrow , be empty or, in case the labels are completely unrelated in either direction, contain .

Referring to Table 1, $addBug(\)$ depends on $addProj(\)$ since we require a project identified by *project_id* in order to add a bug and therefore $1 \rightarrow addProj("ABC" "ERP") \rightarrow 101 \rightarrow addBug(1 "U" "V")$. Similarly $delProj(1) \rightarrow assignProj(1 2)$ as $delProj(\)$ would disable the execution of $assignProj(\)$. Finally, $11 \rightarrow addIssue(1 2 D E) \rightarrow viewProj(1)$ are independent.

Note that, for future reference, we have included on a gray background relations with some labels to be added in the second version of the service. For the third version, where the *addBug* operation is refined, new dependencies between existing operations are underlined and the output of a label is shown by putting the output value before the body of the label with an equal sign e.g. “11 $\rightarrow addIssue(1 2 D E)$ ”. In the same way we highlight new parameters added to the existing signatures.

Having lifted information about conflicts and dependencies to labels, we can use this information in two different ways, for filtering out invalid sequences and for defining equivalence classes. If the observation function σ is faithful, we are able to determine if a sequence of labels s is a valid observation.

- If $l_1 \rightarrow l_2 \in s$ such that $l_1 \rightarrow l_2$, then l_1 precedes l_2 .
- If $l_1 \rightarrow l_2 \in O$ such that $l_1 \rightarrow l_2$ then l_1 precedes l_2 in every sequence s containing l_2 .

Table 1. Asymmetric conflicts and dependencies

First Sec ()	1 addProj ("ABC", "ERP")	2 addUser ("A", "B", "t", "abc")	assignProj (1, 2)	<u>101 addBug</u> ("ABC", "ERP", 1)	... ("U", "V", 2)	<u>11 addIssue</u> ("D", "E")	updttIssSt (1, 11, 2, "XYZ")	dellssSt (11, 22)	delIssue (11)
1 addProj ("ABC", "ERP")					...				
2 addUser ("A", "B", "t", "abc")					-	...			
assignProj (1, 2)						...			
<u>101 addBug</u> ² ("U", "V", 2)						...			
updateBugSt ("ABC", "ERP", 1)						...			
delBugSt ("ABC", "ERP", 1)						...			
delBug ("ABC", "ERP", 1)						...			
unassignProj ("ABC", "ERP", 1)						...			
unassignBug ("ABC", "ERP", 1)						...			
delUser viewUser						...			
updttProj (1, "DEF", "ERP")						...			
updateUsr ("ABC", "ERP", 1)						...			
viewProj (1)						...			
delProj (1)						...			
<u>11 addIssue</u> ³ ("D", "E")						...			
updttIssSt (1, 11, 2, "XYZ")						...			
dellssSt (11, 22)						...			
delIssue (11)						...			

In particular, $l_1 \neq l_2$ and $l_2 \neq l_1$ implies that there is no sequence containing both labels. We denote the set of sequences satisfying these conditions by $O \subseteq O'$. Given a finite approximation of dependency and conflict relations, this feature could be used to filter out test cases that are not executable according to the model.

Moreover, we can partition sequences of labels into equivalence classes, called traces, by considering them equivalent if they differ only for the order of independent labels. The set $Traces(\)$ is the quotient of O under this equivalence. These traces are a generalisation of the classical notion [12] taking into account asymmetric dependencies. Since all sequences in a trace represent the same concurrent behaviour, we can avoid running more than one test from each trace, thus potentially reducing the size of our test suite. However, in this paper we are concerned with the evolution of observable behaviour, not its reduction with respect to a single version of the system.

Example 4 (example of traces through example). With labels as given in Table. 1, a trace [1 addProj("X" "Y"); 2 addUser("A" "B" "t" "m"); assignProj(1 2); viewProj(1)] contains these additional sequences.

² Label updated affecting dependencies.

³ Label updated without affecting dependencies.

```

1 addProj("X "Y );2 addUser("A "B "t "m );assignProj(1 2);viewProj(1)
1 addProj("X "Y );2 addUser("A "B "t "m );viewProj(1);assignProj(1 2)
1 addProj("X "Y );viewProj(1);2 addUser("A "B "t "m );assignProj(1 2)

```

In order to study the effect of evolution of the service on the observable behaviour, we have to consider the changes to dependencies and conflicts on labels. For example, if operations are extended by new features, this will result in more specific pre and post conditions and therefore create new dependencies. But if we introduce new conflicts or dependencies, we will potentially make illegal existing sequences or differentiate between sequences that previously have been equivalent. Where we want to preserve observable behaviour, dependencies and conflicts have to be reflected, i.e., each dependency or conflict in the new version has to be matched by a corresponding one in the old version. This condition for preservation of behaviour at the level of observations has been studied in detail in [9]. In the following section we are going to use it to justify our classification of test cases.

3 Model-Based Evolution

In this section, we first present an evolution scenario in two steps. Then we use the scenario to illustrate our approach to regression test suite reduction.

Evolution Scenarios. In the first evolution step, the Bug Tracker service is extended in order to record Issues with the projects, i.e., concerns raised by users that may not be faults yet indicate deviations from their actual needs. The additional rules and extended type graph are shown in Fig. 4.

In the second evolution step we include a feature to record, among other details, a priority level while adding a bug. That means, the signature of *addBug()* is changed as well as its specification by the rule. Not surprisingly therefore, the modified operation will have additional dependencies, such as *addUser()* → *addBug()*. A minor update to *addIssue()* means that the description of the *status* is preset to “First Report”

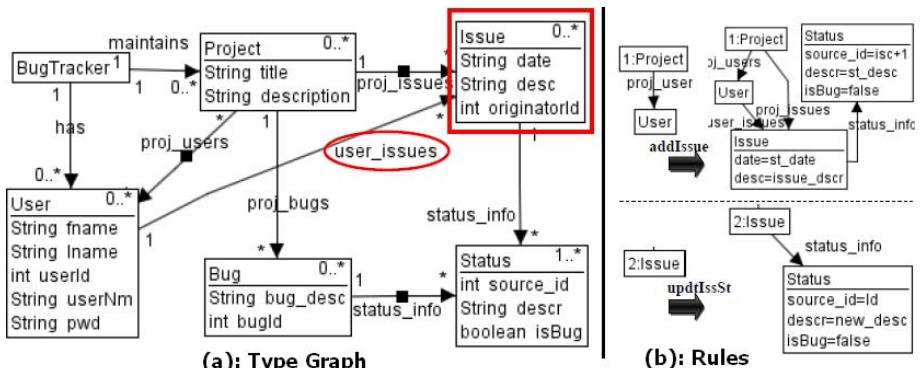


Fig. 4. BugTracker, evolution to Version 2

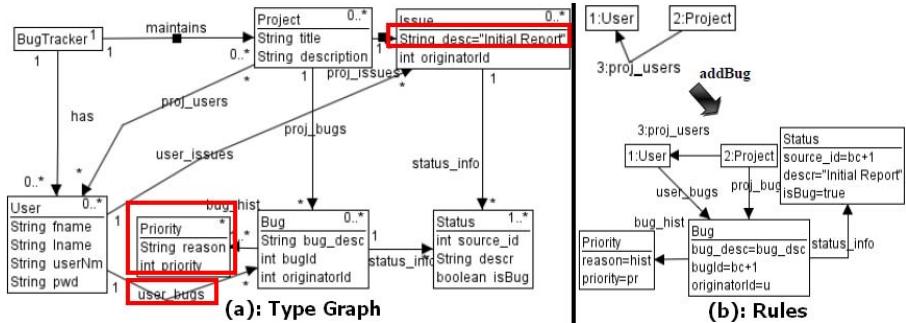


Fig. 5. BugTracker, evolution to Version 3

when the issue is initially reported. There is no change to the signature in this case, and the dependencies and conflicts are not affected. The new version of the changed rule along with the type graph are shown in Fig. 5.

Classification of Test Cases. Given a regression test suite RTS for one version SUT of the system under test, we are going to provide a classification of test cases with respect to an evolution of SUT into SUT' that will distinguish

- *obsolete* test cases OB , that are no longer executable in SUT' , either because signatures have changed or because additional preconditions in the model prevent the execution of operations;
- *reusable* test cases RU , that are still executable in SUT' ;
- *required* test cases RQ , that are still executable and test new or modified functionality in SUT' .

We will refer to the three versions of our model as V1 V2 and V3.

Traces may become *obsolete* because of changes in the operation signatures. In this case, $O \setminus O'$ are labels that are valid for SUT , but invalid in SUT' , e.g., due to missing or incorrectly typed parameters where O' represents the set of labels according to the new version. All traces containing these labels are obsolete as well. In addition, traces could become obsolete because of new dependencies or conflicts emerging in SUT' . The total set of obsolete traces is $O \setminus O'$. To see if a sequence s is obsolete in SUT' we have to check (1) if there are any new dependencies $l_1 \rightarrow l_2$ between labels l_2 in s and labels l_1 not preceding l_2 in s and (2) if there are new conflicts $l_1 \leftrightarrow l_2$ between l_2 in s and l_1 occurring in s after l_2 . If this is not the case, the sequence remains valid and reusable RU .

In the evolution step V1 V2, all conflicts and dependencies are reflected because, while new rules were added, existing rules have not been changed. Hence all traces are preserved and therefore $OB = \emptyset$. For V2 V3, both signature and dependencies have evolved. In particular, $addBug(projId\ bug_desc\ status_desc)$ is obsolete, so all traces containing labels based on this operation are obsolete as well. Instead there are new labels based on the extended signature $addBug(projId\ userId\ priority\ bug_desc\ status_desc)$. We notice that there are new

dependencies shown underlined in Table. 1 which render some of the traces obsolete e.g. $\text{addProj}(\underline{\quad})$; $\text{addBug}(\underline{\quad})$; $\text{viewProj}(\underline{\quad})$ was possible in V2 but not in V3 owing to additional dependency $\text{addUser}(\underline{\quad})$ $\text{addBug}(\underline{\quad})$.

Test cases in RQ , which exercise operations that may have changed or are affected by changes to other operations, are classified as *required*. Denote by $M \subseteq O - O'$ the set of labels such that either their specification or implementation has changed. The set of labels directly and indirectly affected includes M and all labels l_2 such that a label l_1 is affected and $l_1 \rightarrow l_2$ or $l_1 \leftarrow l_2$. The set of required test cases is therefore given by the set of all reusable ones RU which contain at least one affected label.

In evolution $V1 \rightarrow V2$, $RQ = RU$ since there are no modifications to existing operations. New test cases will be required to validate the newly added operations, but this is out of the scope of regression testing. Considering $V2 \rightarrow V3$, we find that $\text{addIssue}(\underline{\quad})$ have been modified and therefore any traces involving their labels are required. Traces containing $\text{addProject}(\underline{\quad})$ and $\text{addUser}(\underline{\quad})$ are required as well because they have dependency relation with $\text{addIssue}(\underline{\quad})$.

4 Evaluation

In this section we evaluate, on a small but real example, both the correctness of our method and the reduction in the set of test cases obtained. That means, we will answer the questions: Do the smaller sets of required test cases RQ find the same faults as the larger sets of reusable test cases RU ? What is the difference in size between RQ and RU and what would be the smallest test set able to find the faults seeded?

For each evolution step the evaluation is performed in four steps that are outlined below and explained in more detail throughout the section.

1. Generation of test cases.
2. Validation of the quality of the entire test suite.
3. Classification of test cases into OB , RU , and RQ .
4. Validation of the quality and required size of RQ by comparing the results of executing RQ and RU .

We generated test cases manually, based on the information in the model, but without applying a formal notion of coverage. The completeness of the test set is evaluated instead through *fault seeding*, i.e., deliberate introduction of faults to be detected by the execution of test cases. The percentage of the seeded faults detected provides a statistical measure of the capability of the test set to find similar errors in the system, i.e., a measure of confidence in our test suite [13]. In order to decide which faults to introduce we identified suitable fault types, and then developed rules for seeding them automatically. After applying the rules to the code of the system, we execute the entire test suite to assess its quality. In an iterative process we add test cases until all of the seeded errors were detected.

After applying to the resulting test set the classification described in Section 3, we validate the completeness of RQ against RU by seeding errors into the classes of our service implementation that were modified in the recent evolution step. We then run the tests in both RQ and RU , comparing their results. The evaluation is based on

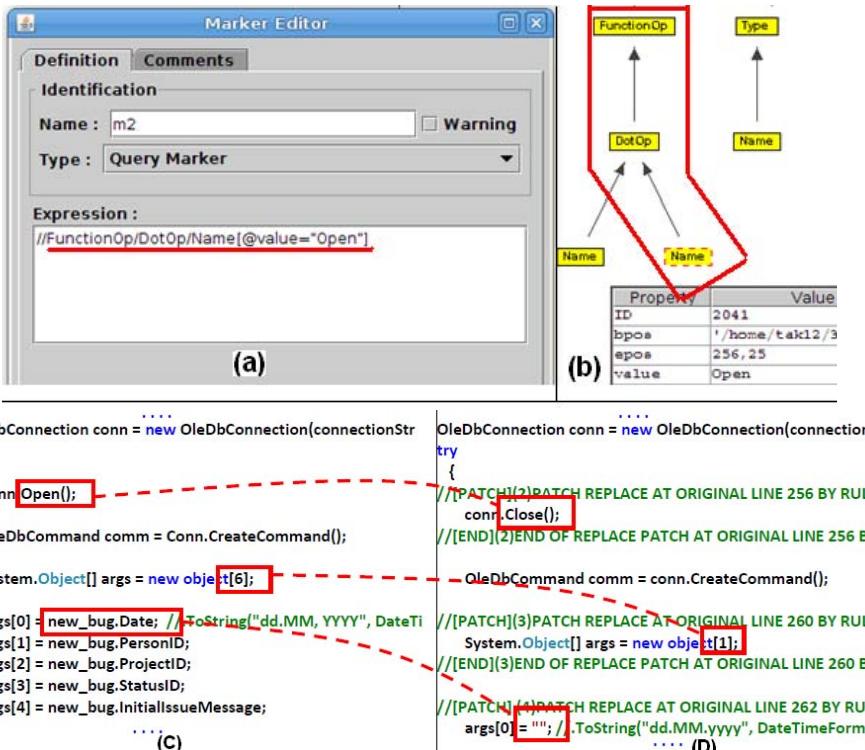


Fig. 6. Fault seeding with L-Care

implementations in C# of the three versions of the Bug Tracker service. The programming environment Pex⁴ has been used for automated unit testing of individual classes. Pex is able to generate test cases based on analysing the source code, with the aim of detecting faults that could lead to runtime errors such as inappropriate exception handling. In our report below we do not include these tests because unit testing is part of the coding at the provider's site while we are concerned with service-level acceptance testing by the client. Therefore, test cases we have generated are concerned with deviations from the public specification of the service interface. We have generated 66 test cases for version V1, 83 test cases for version V2 and 101 test cases for version V3.

Faults are classified by [14], into *domain* and *computation* faults. A domain fault results from control flow errors, where programs follow the wrong path, while a computation fault occurs when the programme delivers incorrect results while following a correct path (usually due to errors in assignments or invocations). More specifically, we have followed the fault types discussed in [15], which also supports calculating a measure of confidence in a test suite. Rules for seeding faults according to these types are implemented in the source code transformation tool L-Care⁵, which allows to define markers based on *XPath* queries as shown in Fig. 6(a) on an XML representation

⁴ p rese r roso o e s pro e s pe

⁵ A product of p e o og es o

Table 2. Distribution of seeded faults

Fault Type	# of Seeded Faults			Code Examples	
	V1	V1	V2	Correct Statement	Mutant Statement
Wrong declaration	6	8	9	new object[6]	new object[0]
Wrong assignment	23	34	35	args[0] DateTime.Now; args[0] “ ”;	
Wrong proc. handling	27	32	35	throw ex	// throw ex
Control faults	22	27	29	if (conn.Open ...)	if (conn.Open ! ...)
I/O faults	27	32	35	conn.Open()	conn.Close()
Total	105	133	143		

of the code. A sketch of this XML in tree form is shown in Fig. 6(b). Examples of the original and the fault-seeded code are shown in Fig. 6 (c) and (d) respectively. Table. 2 shows the total number of faults seeded for each version as well as a breakdown into the different types along with typical representatives.

We tested all the three versions, extending our test suites until all the seeded faults were detected. Our test cases classification was based on computing a conservative (over-)approximation of the actual dependencies and conflicts between labels using the AGG tool [11]. Disregarding the data content, we keep track only of the fact that two parameters in two labels are instantiated with the same value. This reduction is safe because it leads to more, rather than less dependencies and conflicts between concrete labels, and thus to more test cases in *RQ*. In the last step of the evaluation we seed faults in the modified classes of *V2* and *V3* only and execute the two sets of required test cases *RQ* to determine if all of the seeded faults are discovered and how many test cases are actually required to discover them. We have seeded 28 and 18 faults in *V2* and *V3*, respectively, the smaller numbers owing to the size of the changed classes in comparison to the entire code base. The results are reported in Table. 3.

Table 3. Test case classification and success rate

Test cases	V1 V2		V2 V3	
	produced	successful	produced	successful
Obsolete (<i>OB</i>)	0	0	12	0
Reusable (<i>RU</i>)	66	0	45	12
Required (<i>RQ</i>)	0	0	26	12
New (<i>NT</i>)	–	17	–	18

We record the number of test cases in each category *produced* by our classification as well as the number of test cases actually *successful* in finding faults. Of step *V1* – *V2* we recall that *OB* – *RQ* because none of the existing operations were modified. Unsurprisingly, therefore, none of the remaining test cases in *RU* found any fault, but 17 new test cases *NT* had to be produced to detect faults seeded into newly added operations. With the second evolution step, 26 out of 45 existing test cases were classified as required *RQ*, of which 12 were successful in finding faults. Again, 18 new test cases were added to cover features not addressed by existing test cases. That means, our reduction in the size of test suites has not resulted in missing any faults, i.e., the

numbers of faults discovered using RU and RQ are the same. The reduction in size is significant, but probably still not optimal, because a smaller set of 12 rather than 26 test cases would have been sufficient. This is despite an exhaustive error seeding strategy, which produced faults of the designated types wherever this was possible in the code. The reason could be in over approximation of dependencies and conflicts which, like in many static analysis approaches, leads us to err on the captious side.

To conclude the evaluation, let us discuss a possible threat to the validity of these results. When using the set of reusable test cases RU as a benchmark for the required ones RQ , the assessment depends on the quality of the original test suite, which was evaluated by fault seeding. But fault seeding will only deliver relevant results for the types of faults actually sown, while unexpected or unusual faults are not considered. Our approach here was to use approaches to fault classification from the literature, but in order to gather relevant statistics about the costs savings possible we would require data on error distributions from real projects.

5 Related Work

Several techniques [16,17,18] have been using model level information for regression testing. Extended finite state machine (EFSM) are considered in [16], where interaction patterns between functional elements represented by transitions are used for test set reduction. Two tests are considered equivalent if they represent the same interaction pattern. Therefore, whenever a transition is added or deleted, the effect of the model on the transition, the effect of the transition on the model and any side effects are tested for. That means test cases are selected with respect to elementary modifications of the state machine model .

EFSM are also considered in [17] where a set of elementary modifications EM is identified. Two types of dependencies, data dependencies DD and control dependencies CD are discussed. A state dependence graph SDG represents DD and CD visually and a change in the SDG leads to a regression testing requirement to verify the effect of the modification.

The technique presented in [18] uses UML use case and class diagrams with operations described by pre and post conditions in OCL. A unique sequence diagram is associated with a use case to specify all possible object interactions realising the use case. Changes in the model are identified by comparing their XMI representations.

An approach to regression testing of web services suggested by [2] makes use of unit tests based on JUnit. Test cases are produced by the developer, who generates QoS assertions and XML-encoded test suites and monitors I/O data of previous test logs to see if the behaviour is changed.

[19] constructs a global control flow graph CFG and defines special call nodes for each remote service invocation. A CFG containing a call node, referred to as non-terminal graph, is converted to a terminal graph by inserting the CFG corresponding to that call node. Whenever an operation is modified, the previous and the resulting call graphs are compared to find the differences and all downstream edges are marked as “dangerous” once a modified node is marked.

We make use of semantic information in service interfaces and lift dependencies and conflicts derived to the level of observable actions as they would be seen by a user

of the service. Apart from differences in the models used (visual contracts instead of state machines, sequence diagrams or OCL) we employ (asymmetric) dependencies as well as conflicts to characterise admissible sequences of observations. The use of asymmetric relations is due to our richer notion of model, which accounts for data transformation rather than automata-like protocols the order or frequency of method invocations. Dependency information used, e.g., in [16,17] is instead derived from state machines. Pre and post conditions on application data are also used with [18]. While conceptually close, our visual contracts are more easily usable than a textual encoding in OCL and provide a formal operational semantics with a well-established theory of concurrency as a basis for verifying formally the correctness of our approach.

6 Conclusion and Outlook

In this paper we have presented a method to reduce the size of a regression test suite based on an analysis of the dependencies and conflicts between visual contracts specifying the preconditions and effects of operations. The method is applicable to all software systems that have interfaces specified in this way, but is particularly relevant for services because of the lack of access to implementation code and the potential cost involved in running a large number of tests through a remote and potentially payable provider. The method is backed up conceptually and formally by a related paper [9] providing an observational view of the concurrent behaviour of graph transformation systems. In the present paper we have evaluated the approach through the development of a case study showing that (1) the reduced test sets could find all the faults detected by the larger sets while (2) being significantly smaller.

As future work we are aiming to automate the generation of dependencies and conflicts on labels, formalising the over approximation required to represent finitely a relation on an infinite set of labels. We are also working on coverage criteria for test suites based on contract dependencies as well as on a solution to reduce test suites by omitting (the generation of) equivalent test sequences.

References

1. Canfora, G., Penta, M.D.: Testing services and service-centric systems: Challenges and opportunities. *IT Professional* 8, 10–17 (2006)
2. Penta, M., Bruno, M., Esposito, G., Mazza, V., Canfora, G.: Web services regression testing. *Test and Analysis of Web Services*, 205–234 (2007)
3. Rothermel, G., Harrold, M.J.: Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering* 22 (1996)
4. Leung, H., White, L.: Insights into regression testing [software testing]. In: Proc. Conference on Software Maintenance, pp. 60–69 (October 1989)
5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2006)
6. Heckel, R.: Graph transformation in a nutshell. In: *Electr. Notes Theor. Comput. Sci.*, pp. 187–198. Elsevier, Amsterdam (2006)

7. Lohmann, M., Sauer, S., Engels, G.: Executable visual contracts. In: VLHCC 2005: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 63–70. IEEE Computer Society, Washington, DC, USA (2005)
8. Lohmann, M., Mariani, L., Heckel, R.: A model-driven approach to discovery, testing and monitoring of web services. Test and Analysis of Web Services, 173–204 (2007)
9. Khan, T., Machado, R., Heckel, R.: On the observable behavior of graph transformation systems. Technical Report CS-10-003, Department of Computer Sciences (August 2010),
p s e peop e o serv e p
10. Baldan, P., Corradini, A., Montanari, U.: Contextual petri nets, asymmetric event structures, and processes. Information and Computation 171(1), 1–49 (2001)
11. AGG: AGG - Attributed Graph Grammar System Environment (2007),
p s s er e gg
12. Diekert, V., Rozenberg, G.: The Book of Traces. World Scientific Publishing Co., Inc., River Edge (1995)
13. Pfeleger, S.L.: Software Engineering: Theory and Practice. Prentice Hall PTR, Upper Saddle River (2001)
14. Howden, W.: Reliability of the path analysis testing strategy. IEEE Transactions on Software Engineering SE-2(3), 208–215 (1976)
15. Pasquini, A., Agostino, E.D.: Fault seeding for software reliability model validation. Control Engineering Practice 3(7), 993–999 (1995)
16. Korel, B., Tahat, L., Vaysburg, B.: Model based regression test reduction using dependence analysis. In: Proc. Conference on Software Maintenance, pp. 214–223 (2002)
17. Chen, Y., Probert, R.L., Ural, H.: Model-based regression test suite generation using dependence analysis. In: A-MOST 2007: Proc. of the 3rd Intl. Workshop on Advances in Model-based Testing, pp. 54–62. ACM, New York (2007)
18. Briand, L.C., Labiche, Y., He, S.: Automating regression test selection based on UML designs. Inf. Softw. Technol. 51(1), 16–30 (2009)
19. Ruth, M., Oh, S., Loup, A., Horton, B., Gallet, O., Mata, M., Tu, S.: Towards automatic regression test selection for web services. In: COMPSAC 2007: Proceedings of the 31st Annual International Computer Software and Applications Conference, pp. 729–736. IEEE Computer Society, Washington, DC, USA (2007)

Incremental Clone Detection and Elimination for Erlang Programs

Huiqing Li and Simon Thompson

School of Computing, University of Kent, UK
`{H.Li, S.J.Thompson}@kent.ac.uk`

Abstract. A well-known bad code smell in refactoring and software maintenance is the existence of *code clones*, which are code fragments that are identical or similar to one another. This paper describes an approach to *incrementally* detecting ‘similar’ code based on the notion of least-general common abstraction, or *anti-unification*, as well as a framework for user-controlled incremental elimination of code clones within the context of Erlang programs. The clone detection algorithm proposed in this paper achieves 100% precision, high recall rate, and is user-customisable regarding the granularity of the clone classes reported. By detecting and eliminating clones in an incremental way, we make it possible for the tool to be used in an interactive way even with large codebases. Both the clone detection and elimination functionalities are integrated with Wrangler, a tool for interactive refactoring of Erlang programs. We evaluate the approach with various case studies.

Keywords: Software maintenance, Refactoring, Code clone detection, Erlang, Program analysis, Program transformation, Erlang, Wrangler.

1 Introduction

Duplicated code, or the existence of code clones, is one of the well-known bad ‘code smells’ when refactoring and software maintenance is concerned. The term ‘duplicated code’, in general, refers to program fragments that are identical or similar to one another; the exact meaning of ‘similar code’ might be substantially different between different application contexts.

The most obvious reason for code duplication is the reuse of existing code, typically by a sequence of *copy*, *paste* and *modify* actions. Duplicated code introduced in this way often indicates program design problems such as a lack of encapsulation or abstraction. This kind of design problem can be corrected by refactoring out the existing clones at a later stage, but could also be avoided by first refactoring then reuse the existing code. In the last decade, substantial research effort has been put into the detection and removal of clones from software systems; however, few such practical tools are available for functional programming languages. The work reported here is of particular value both to the working programmer and the project manager of a larger programming project, in that it allows clone detection to contribute to the ‘dashboard’ reports from incremental nightly builds, for instance.

This paper describes an approach to incrementally detecting ‘similar code’ in Erlang programs based on the notion of *least-general common abstraction*, or *anti-unification* [1,2], as well as a mechanism for incremental automatic clone elimination under the user’s control. We take Erlang as the target language due to our research context. While the implementation discussed in this paper is specific to Erlang’s syntax and static semantics rules; the methodology used by the approach is applicable to other functional programming languages as well.

In general, we say two expressions or expression sequences, A and B, are *similar* if there exists a non-trivial least-general common abstraction, or anti-unifier, C, and two substitutions σ_A and σ_B which take C to A and B respectively. By ‘non-trivial’ we mainly mean that the size of the least-general common abstraction should be above some threshold, but other conditions, such as the complexity of the substitution, can also be specified.

The approach presented in this paper is able, for example, to spot that the two expressions $(X+3)+4$ and $4+(5-(3*X))$ are similar as they are both instances of the expression $Y+Z$, and so both instances of the function

$$\text{add}(Y, Z) \rightarrow Y+Z.$$

When support for clone elimination is one of the major purposes served by a clone detection tool, accuracy and efficiency of the tool are essential for it to be usable in practice.

To achieve a 100% precision rate, i.e., only genuine clones are reported, our approach uses as the representation of an Erlang program the Abstract Syntax Tree (AST) for the parsed program annotated with static semantic information. Syntactic and static semantic information together make it possible that only those genuine, and syntactically well-formed, clones are reported to the user.

Scalability and efficiency, the major challenges faced by AST and/or semantics based clone detection approaches, are achieved by an *incremental* two-phase clone detection technique. The first phase uses a more efficient, but less accurate, syntactic technique to identify candidates which might be clones; the initial result is then assessed by means of an AST and static semantics based analysis during the second phase to give only genuine clones. When part of the codebase has been changed, the original clone detection result is no longer up-to-date, due to either the change of locations or the textual change of code. To keep the clone report up-to-date, instead of re-running the clone detection from scratch, the *incremental* clone detection algorithm reuses and updates the data collected from the previous run of the clone detection, and only processes clones related to the code that has been modified, added or deleted.

Our clone detection tool reports clone classes. As shown in the lower window of Fig. 1, each clone class is a set of code fragments in which any two of the code fragments are identical or similar to each other. Each clone class is associated with the least-general abstraction in the format of a function abstraction named `new_fun`. Variable names of the form `NewVar_i` are generated by the clone detector. For each member of a clone class, the clone detector also generates an application instance of `new_fun`, which is the function call that arises from unifying the class member with the function definition represented by `new_fun`.

The screenshot shows the Wrangler Erlang Shell interface. The title bar reads "emacs@HL-LT". The menu bar includes File, Edit, Options, Buffers, Tools, Refactor, Inspector, Erlang, and Help. Below the menu is a toolbar with icons for file operations like Open, Save, Print, and Cut/Paste.

The main area displays two Erlang code snippets side-by-side:

```

loop_a() ->
    receive
        stop -> ok;
        {msg,_Msg,0} -> loop_a();
        {msg,Msg,N} ->
            io:format("ping!~n"),
            timer:sleep(500),
            b!(msg,Msg,N-1),
            loop_a()
    after
        15000 ->
            io:format("Ping got bored, "
                      "exiting.~n"),
-->*- pingpong.erl 50% (42,20) (Erlang)
-->*- pingpong.erl 75% (60,0) (Erlang)

```

On the right side of the interface, there is a vertical scroll bar and a status bar at the bottom.

Below the code snippets, the text "Similar Code Detection Results Sorted by Code Size." is displayed. The results show two clones found in the file pingpong.erl:

- Clone 1.** This code appears twice:
 - [c:/cygwin/home/h1/test/pingpong.erl:55.12-57.27:](#) new_fun(Msg, N, "pong!~n", a)
 - [c:/cygwin/home/h1/test/pingpong.erl:39.12-41.27:](#) new_fun(Msg, N, "ping!~n", b)
- The cloned expression/function after generalisation:**

```

new_fun(Msg, N, NewVar_1, NewVar_2) ->
    io:format(NewVar_1),
    timer:sleep(500),
    NewVar_2!(msg,Msg,N-1).

```

Clone 2. This code appears twice:

- [c:/cygwin/home/h1/test/pingpong.erl:56.12-58.20:](#) new_fun(Msg, N, a, fun () -> loop_b() end)
- [c:/cygwin/home/h1/test/pingpong.erl:40.12-42.20:](#) new_fun(Msg, N, b, fun () -> loop_a() end)

-1*- *Wrangler-Erl-Shell* 50% (31,0) (Comint:run Compilation)

Fig. 1. A snapshot showing similar code detection with Wrangler

One aim of clone detection is to identify them so that they can be eliminated. The general approach to removing a cloned code fragment in the functional programming paradigm is to replace it with a function call to the least-general common abstraction of the clone class to which the code fragment belongs. In theory, it is possible to eliminate all clones found fully automatically without control from the user, however in practice, this is not a desirable way for various reasons. Instead of eliminating clones fully automatically, our framework allows the user to eliminate clones in an *incremental* way. We automate things that should be automated, while allowing the user to have control over the clone elimination process when it is necessary.

A non-incremental similar code detection and elimination framework was first added to Wrangler in 2009 [3]. The contribution of this paper is to provide:

- an incremental algorithm which works substantially more efficiently than the original standalone algorithm for larger projects in analyses;
- a framework that gives the user fine control of the granularity of the clones reported and a clear vision of the code after eliminating a specific clone;
- a method for tracking the evolution of clones during software lifetime, and
- a mechanism for clone detection to be included in the standard workflow and reporting of projects subject to continuous integration or regular builds.

The remainder of the paper is organised as follows. Section 2 gives an overview of Erlang and the refactoring tool Wrangler. Section 3 introduces some terminology to be used in this paper; Section 4 describes the incremental clone detection algorithm. The elimination of code clones is discussed in Section 5, and an evaluation of the tool is given in Section 6. Section 7 gives an overview of related work, and Section 8 concludes the paper and briefly discusses future work.

2 Erlang and Wrangler

Erlang [4] is a strict, impure, dynamically typed functional programming language with support for higher-order functions, pattern matching, concurrency, communication, distribution, fault-tolerance, and dynamic code loading. Erlang allows static scoping of variables, in other words, matching a variable to its binding only requires analysis of the program text, however some variable scoping rules in Erlang are rather different from other functional programming languages.

An Erlang program typically consists of a number of modules, each of which defines a collection of functions. Only functions exported explicitly through the `export` directive may be called from other modules. In Erlang, a function name can be defined with different arities, and the same function name with different arities can represent entirely different functions computationally.

Wrangler [5] is a tool that supports interactive refactoring of Erlang programs. It is integrated with (X)Emacs, as shown in Fig. 1, and as well as with Eclipse through ErlIDE. Wrangler is implemented in Erlang, and downloadable from <http://www.cs.kent.ac.uk/projects/wrangler/Home.html>.

3 Terminology

Anti-unification and Unification. Anti-unification, first proposed in 1970 by Plotkin [1] and Reynolds [2], applies the process of *generalisation* to pairs, or sets, of terms. The resulting term captures all the commonalities of the input terms. Given terms E_1, \dots, E_n , we say that E is a *generalisation* of E_1, \dots, E_n if there exist substitutions σ_i for each $E_i, 1 \leq i \leq n$, such that $E_i = E\sigma_i$. E is a *least-general* common generalisation of E_1, \dots, E_n if for each E' which is also a common generalisation of E_1, \dots, E_n , there exists a substitution θ such that $E = E'\theta$; it is not difficult to see that these are unique (up to renaming of variables) and so we call any of them the least-general common generalisation. The least-general common generalisation of E_1, \dots, E_n is called the *anti-unifier* of E_1, \dots, E_n , and the process of finding the anti-unifier is called *anti-unification*.

To apply anti-unification techniques to ASTs of Erlang programs, restrictions as to which kinds of subtrees can be replaced by a variable, and which cannot, need to be taken into account. For instance, objects of certain syntactic categories, such as operators, guard expressions, record names, cannot be abstracted and passed in as the values of function parameters, and therefore should not be replaced by a variable. Furthermore, an AST subtree which exports some of its locally declared variables should not be replaced by a variable either; whereas it

is fine to substitute the function name in a function application with a variable because higher order functions are supported by Erlang.

Unification, on the other hand, is the process of finding substitutions of terms for variables to make expressions identical [6]. The unification technique forms the basis of the clone elimination process.

Similarity Score. Anti-unification provides a concrete way of measuring the structural similarity between terms by showing how both terms can be made equal. In order to measure the similarity between terms in a quantitative way, we defined the *similarity score* between terms.

Let E be the anti-unifier of sub-trees E_1, \dots, E_n , the similarity score of E_1, \dots, E_n is computed by the following formula: $\min\{S_E/S_{E_1}, \dots, S_E/S_{E_n}\}$, where $S_E, S_{E_1} \dots S_{E_n}$ represent the number of nodes in $E, E_1 \dots E_n$ respectively. The similarity score allows the user to specify how similar two sub-trees should be to be considered as clones. Given a similarity score as the threshold, we say that a set of sub-trees are *similar* if their similarity score is above the threshold.

Clone Classes. A *clone class* is a set of code fragments in which any two of the code fragments are identical or similar to each other. In the context of this paper, each member of a clone class is a sequence of Erlang expressions. We say a clone class C is maximal if there does not exist a clone class C' such that $|C| \leq |C'|$, and for each class member, E_i say, of C , there exists a clone member, E_j say, of C' , such that E_i is a proper sub-sequence of E_j . Only those maximal clone classes are reported by our clone detection tool.

4 The Clone Detection Algorithm

The similar code detector takes a project (that is, a set of Erlang modules), and a set of parameters as input, performs clone detection, and reports the clone classes found. The tool is integrated with an IDE (Emacs or Eclipse), but it can also be run from the command line. Five parameters need to be specified; if no value is supplied a suitable default value is used. The parameters are:

- the minimum number of expressions included in a clone which is a sequence of expressions, denoted by E_{min} ;
- the minimum number of lexical tokens included in a clone, denoted by T_{min} ;
- the maximum number of new parameters of the least-general common abstraction function, denoted by P_{max} ;
- the minimum number of class members of a clone class, denoted by F_{min} ;
- the similarity score threshold, denoted by $SimiScore$.

With these parameters, the user can have a fine control of the granularity of the clone classes reported. For example, to make the tool only report identical code fragments, the user just need to set the value of P_{max} to 0.

As shown in Fig. 1, each clone class is reported by giving the number of instances of the cloned code, the least-general common generalisation in the

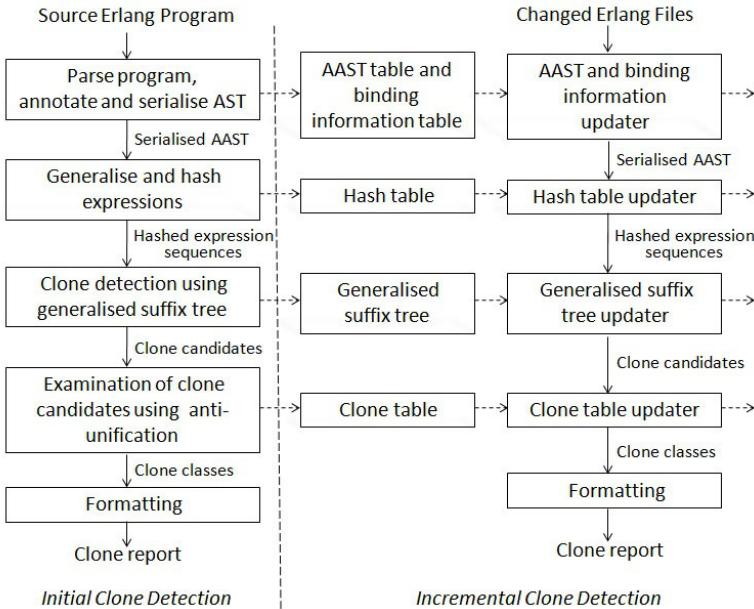


Fig. 2. An Overview of the Incremental Clone Detection Process

format of a function abstraction named `new_fun`, each clone instance's start and end locations, as well as the application instance of `new_fun`, which is the function call that arises from unifying the class member with function definition represented by `new_fun`. Scalability is tackled from two aspects:

- A two-phase clone detection technique is used. The first phase carries out a quick, semantics-unaware clone detection over a generalised version of the program, and reports initial clone candidates; the second phase examines the initial clone candidates in the context of the original program by means of anti-unification, getting rid of false positives;
- Incremental update of information collected from the various stages of the previous run of the clone detection tool when program code has been changed.

An overview of the algorithm is shown in Fig. 2. Next, we first describe the initial clone detection algorithm, then the incremental part.

4.1 The Initial Detection Algorithm

The initial clone detection algorithm is an extended and adapted version of the standalone algorithm presented in [3]. While both follow the same steps, the algorithm presented here is designed to make incremental clone detection possible, and provides better usability. We explain it in more detail now.

Parse Program, annotate and serialise AST. Erlang files are lexed and parsed into ASTs. In order to reflect the original program text, the Erlang pre-processor is bypassed to avoid macro expansion, file inclusion, conditional compilation, etc. Both line and column numbers of identifiers are kept in the ASTs generated, since location information is used to map between different representations of the same piece of code in the source. Binding information of variables and function names, which is needed during the anti_unification process, is annotated to the AST in terms of defining and use locations.

Location information is needed by the clone detector, but absolute locations are sensitive to changes, i.e. a change made to a particular place of a file could possibly affect all the locations following it. With incremental clone detection in mind, we choose to use relative locations to identify program entities, while recording each function’s actual starting line in the file. With relative locations, every function starts from line 1 at column 1. The benefit of this is that we can ensure pure location change does not affect the initial clone candidate result, and all new clone candidates introduced are due to structural change.

The AAST representation of each function is then traversed, and expression sequences are collected. In this way, each function is mapped into a list of expression sequences. The AAST representation of each expression statement is stored in an ETS (Erlang Term Storage) table for use by the later stages of the algorithm and the incremental clone detection. To reduce the time spending on AAST traversal while trying to locate a specific syntax phrase in the AAST, and also to reduce the time on AAST updating when incremental clone detection is concerned, each object in the ETS table represents the AAST of a single expression statement, instead of the AAST of a whole Erlang file.

Generalise and Hash Expressions. This step takes an expression sequence generated from the previous step a time, each expression statement in the sequence is first structurally generalised, then pretty-printed and hashed into an integer value. Only expression statements that share the same generalised form get the same hash value. Therefore, we map each expression sequence into a sequence of integers.

The aim of structural generalisation is to capture as much structural similarity between expressions as possible while keeping each expression’s original structural skeleton. This process traverses each expression statement subtree in a top-down order, and replaces certain kinds of subtrees with a single node representing a placeholder. A subtree is replaced by a placeholder only if syntactically it is legal to replace that subtree with a node representing a variable, and the subtree does not represent a pattern, a match expression or a compound expression such as a `receive` expression, a `try...catch` expression, etc.

Initial Clone Detection using a Generalised Suffix Tree. This step turns each integer sequence generated from the previous step into a string, and builds a generalised suffix tree on the strings generated. Cloned strings are then collected

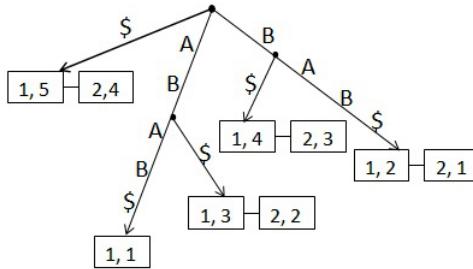


Fig. 3. A generalised suffix tree for two strings ABAB\$ and BAB\$

from the suffix tree, and each group of cloned strings is then mapped back to a group of expression sequences which share the same generalised representation.

Suffix tree analysis [7] is the technique used by most text or token-based clone detection approaches because of its speed [8,9]. A suffix tree is a representation of a single string as a tree where every suffix is represented by a path from the root to a leaf. The edges are labelled with the substrings, and paths with common prefixes share an edge. A generalised suffix tree is a suffix tree that represents the suffix of a set of strings. Fig. 3 shows an example of the generalised suffix tree representation of two strings ABAB\$ and BAB\$. The numbers in the leaf nodes are string number and starting position of the suffix in the string. We use generalised suffix tree instead of the standard suffix tree as in [3] for two reasons:

- The standard suffix tree algorithm accepts only a single string as input. To build a suffix tree over a collection of strings, we will have to concatenate all the strings into a single one, and then build a suffix tree over the concatenated string. As a result, clone strings might actually come from two or more separated strings, and therefore need to be further processed;
- With generalised suffix tree, a string can be removed from, or inserted into, the tree easily, which is exactly what we need for incremental clone detection.

Examine Clone Candidates using Anti-unification. The previous step returns a collection of clone classes whose class members are structurally similar, but do not necessarily share a non-trivial anti-unifier; even so it helps to reduce the amount of comparisons needed significantly. This step examines the initial clone class candidates one by one using anti-unification and returns those genuine clone classes. More details follow.

Generation of clone classes from clone candidates. For each clone class candidate, C say, the clone detector takes a class member, A say, and tries pairwise anti-unification with each of the other class members. The anti-unification result is then analysed and processed to derived clone classes that satisfy all the parameters specified by the user. In order to achieve a high recall rate, when two expression sequences do not anti-unify as a whole, their sub-sequences are also examined. If the whole clone class candidate does not form a real clone class, another class member is selected from the remaining members of C , and the

process is repeated until no more new clone classes can be found. As a result, it is possible to derive none, one, or more clone classes from a single clone candidate. For example, from a clone class candidate with three expression sequences like $\{E_{11}E_{12}E_{13}, E_{21}E_{22}E_{23}, E_{31}E_{32}E_{33}\}$, it is entirely possible to derive three clone classes like: $\{E_{11}E_{12}, E_{21}E_{22}, E_{31}E_{32}\}$, $\{E_{11}E_{12}E_{13}, E_{21}E_{22}E_{23}\}$ and $\{E_{21}E_{22}E_{23}, E_{31}E_{32}E_{33}\}$.

Generation of anti-unifier. The anti-unifier generator takes a clone class and the substitutions inferred during the anti-unification process as input, generalises the expression sequence represented by the first clone class instance over those sub-expressions which are not common to all of the clone instances by replacing those sub-expressions with new variables automatically generated by the clone detector. To ensure that only the minimum number of new parameters are generated, the anti-unifier generator needs to check the static semantics of the sub-expressions to be generalised over, and their corresponding sub-expressions in remaining class instances, so that sub-expressions with the same static semantics and same substitutions are represented by the same new variable. Variables declared by a cloned code fragment in a clone class might be used by the code following it, and the union of these variables should be returned by the anti-unifier so that those variables are still visible to the code following it when the cloned fragment is replaced by an application instance of the anti-unifier.

Generation of application instances. Given the anti-unifier of a clone class and a particular instance of the clone class, the application instance is generated through the unification of the anti-unifier, represented as a function definition named `new_fun`, and the class instance. The application instance gives the user a clear vision of what a cloned code fragment will be replaced with by clone elimination, and therefore helps the user decide whether this clone instance should be eliminated or not. For example, if some of the parameters of the application instance are too complex, the user might want to refactor the code first, then eliminate the clone.

Formatting. Final clone classes are sorted and displayed in three different orders: by the number of duplications, by the size of clone class instances, and by the ranking score of each clone class. The ranking score of a clone class is calculated based on three parameters: the number of parameters of the anti-unifier, the number of terms returned by anti_unifier, and size of the anti-unifier body. Given a clone class anti_unifier, suppose the above three parameters are represented by P , V and L respectively, the ranking score is calculated as: $L/(L + P + V)$.

4.2 The Incremental Detection Algorithm

A change made to a file could affect the existing clone results in two different ways. A structural change could introduce new clone classes, or invalidate some existing clone classes; a location change on the other hand could make the location information of some clone class members out-of-date. Obviously, a

structural change to one part of the file is generally accompanied by location changes to the code following it in the file.

To incrementally update the clone result after changes have been made to the program source, our algorithm reuses and updates the intermediate results returned from the previous run of the clone detection as shown on right-hand side of the diagram in Fig. 2.

The algorithm takes a function as a unit to track the changes made to the program source. For a function that is removed, added, or structurally changed, we have no other choices but to remove/add/update the entities associated with it; whereas for a function with only a location change, only its absolute starting location is updated. Taking a function, instead of a file as the unit for tracking changes, we are able to reuse existing results as much as possible. Next we explain in more detail the various intermediate results that are reused and updated during the incremental clone detection phase.

The AAST Table. The AAST table stores the AAST representation of each expression statement. Each entry of the table is a tuple. The first element of the tuple, which serves as the key, is of the format: {ModuleName, FunctionName, Arity, ExprIndex}, where ModuleName, FunctionName, and Arity together identify a function, and ExprIndex is used to identify an expression statement of this function; the second element of the tuple is the AAST representation of the expression statement, together with the absolute starting line number of the function to which the expression belongs. The AAST updater checks which part of the program has been changed since the last run of the clone detection, and updates the AAST table accordingly. For a function that is deleted, added or modified structurally, the entries associated with that function in the AAST table are also deleted, added, or updated. Only the starting line number is updated if a function only has its location changed.

The Binding Information Table. This table stores the binding structure information for variables of each function. Each entry of the table is a tuple with the first element identifying a function, and the second element representing the binding structure in terms of defining and use locations. This table is updated only if a function has been deleted, added or structurally modified.

The Expression Hash Table. This table stores the mapping from an expression statement to its hash value, therefore from an expression sequence to a sequence of hash values, as well some meta-information about the expression, including the number of lexical tokens, location information, and a boolean flag indicating whether the expression is new. Each expression statement in this table is also identified by a tuple, whose first element is an integer uniquely identifying the expression sequence to which the expression statement belongs, and the second element is a four-element tuple {ModuleName, FunctionName, Arity, ExprIndex} identifying the particular expression.

The Generalised Suffix Tree. The reuse of the generalised suffix tree allows us to avoid re-building the suffix tree for the whole program from scratch. The generalised suffix tree is updated by removing those strings deleted/changed from

it, and by adding the new strings into it. To avoid the re-calculation of clones from the suffix tree, we annotate each internal node with the clone information represented by that node, which is also updated accordingly.

The Clone Table. This table stores the mapping between each initial clone candidate and the clone classes derived from it. Each entry is a tuple whose first element is the clone candidate, and second element is the clone classes derived from it. A clone candidate is only processed if it does not belong to this table, and the result is then added to the table. By tracking changes of the clone table, we are able to track the evolution of clones during software development.

5 Support for Clone Elimination

Working within the functional programming paradigm, the general approach to removing a cloned code fragment is to replace it with an application of a function whose definition represents an abstraction of the cloned code fragment. In theory, it is possible to eliminate all clones found fully automatically, however, our experience [10] shows that this is undesirable for the following reasons:

- Some cloned code fragments logically do not represent a clearly defined functionality; or a cloned code fragment might contain code that logically belongs to the code before, or after, it in the program source.
- The least-general common abstraction generated by the clone detector has to be given a proper name to reflect its functionality; and the parameters of the least-general common abstraction might need to be renamed or re-ordered.
- In the case that a clone class contains code fragments from multiple modules, a proper module has to be selected as the host module of the least-general common abstraction to avoid introducing bad modularity smells.
- Fully automatic clone detection could introduce too many changes to the code base in one go, and makes it harder for the user to follow.

Our clone elimination framework tries to automate things that should be automated, while allowing the user to have control over the clone elimination process when it is necessary. With this framework, the following steps can be followed to eliminate some, or all, cloned code fragments from a given clone class.

1. Copy and paste the anti-unifier of the clone class into an Erlang module;
2. rename variable names if necessary;
3. re-order the function parameters if necessary;
4. rename the function to some suitable name;
5. apply ‘fold expressions against a function definition’ to the new function.

Renaming, reordering of function parameters and folding expressions against a function definition are all refactorings supported by Wrangler. *Folding* is the refactoring which actually removes code clones. It searches for, and highlights, clone instances of the function clause selected, and replaces those clone instances which the user chooses to eliminate with application instances of the function

selected. This refactoring carries out its own clone instance search, and can therefore be applied independently of the clone detection process.

Given a clone report containing a list of clone classes, the user has a number of decisions to make as to which clone classes, or even which instances of a specific clone class, to eliminate. We believe that by showing the least-general common abstraction of each clone class and the application instance associated with each clone class instance, we make this process much easier.

6 Experimental Evaluation

The tool has been applied to various Erlang application code and testing code. In this paper, we take three codebases to evaluate the efficiency and accuracy of the approach. The first codebase is Wrangler itself; the second codebase is an Erlang test suite written with Erlang’s Common Test framework from a mobile industry; and the third codebase includes the application and testing code of both the Erlang compiler and the Erlang stdlib. The experiments were conducted on a laptop with Intel(R) 2.27 GHz processor, 4.00 GB RAM, and running Windows 7. The tool is evaluated in two criteria: runtime efficiency and the number of clones detected. To contrast the performance, we run both the incremental and the standalone clone detection for each codebase and version.

The default parameter setting for the tool, i.e. 5 for E_{min} , 40 for T_{min} , 4 for P_{max} , 2 for F_{min} and 0.8 for $SimiScore$, was used throughout the experiments. The experimental results are shown in Table 1. The first column of the table shows the codebases and their versions we used. For both Wrangler and the Erlang compiler/stdlib, the version numbers are the release numbers, therefore

Table 1. Incremental vs. Standalone Clone Detection

Wrangler	KLOC	Files Changed	Incremental		Standalone	
			Time	Clones	Time	Clones
0.8.7	42.5	70/70	15	18	15	18
0.8.8	44.2	59/80	10	21	15	21
0.8.9	47.4	44/83	8	26	16	26
0.9.0	48.0	9/84	2	26	16	26
0.9.1	48.1	3/84	3	26	17	26
Test Suite						
V0	24.0	26/26	560	371	560	371
V1	23.9	3/26	90	361	550	361
V2	23.9	1/26	54	357	550	357
V3	23.8	2/26	90	346	550	346
V4	23.7	2/26	80	338	540	338
Erlang						
R13B-03	244.3	306/306	94	78	94	78
R13B-04	245.5	71/311	36	79	97	79
R14A	250.8	108/327	40	82	95	82
R14B	251.9	39/321	28	81	94	81

the amount of changes made between two consecutive versions could be large; for the test suite, we use the original test suite as version V0, and each version following represents the codebase after some clones have been eliminated. The second column shows the size of each version of codebase in terms of the number of lines of code. The third column shows the number of files that are changed since the previous version out of the total number of files. The time is measured in seconds, and the clones are measured by the number of clone classes reported.

It is obvious from this table that the processing time can be reduced significantly especially when the amount of changes made between two consecutive versions is small. The tool performance is affected by both the size of the code and the number/size of the initial clone candidates detected. Table 1 shows that the processing time for the test suite is considerably long compared to the other two codebases. This is due to the large amount of clones, and clone candidates, detected. For example, the clone report for the test suite V0 says that the longest clone consists of 86 lines of code, and is duplicated twice; and the most frequently cloned code consists of 5 lines of code, and is duplicated 83 times.

The precision of the tool should be 100% due to the use of static semantics aware analysis during the clone candidate examination phase, and this has been verified through various case studies during the development of the tool. Any false positives reported simply imply a bug in the tool implementation. As to the recall rate, given a set of parameter settings, our tool in theory should be able find all those classes whose members do not textually overlap; but because it is practically impossible to examine this manually with large codebases, and there are no other clone detection tools for Erlang which we can use for comparison, at this stage we cannot give a concrete recall rate for the results reported here.

Compared to other clone detection tools, our tool takes precision and usability of the tool as the top priority. A recent study [11] has shown that up to 75% of clones detected by state-of-the-art tools are false positives, and this has hindered the adoption of clone detection techniques by software developers, no matter how fast the tool is and how large of a code base the tool can work with, as inspection of false positives is a waste of developer time.

7 Related Work

A survey by Roy et. al. provides a qualitative comparison and evaluation of the current state-of-the-art in clone detection techniques and tools [12]. Overall there are text-based approaches [13,14], token-based approaches [8,15], AST-based approaches [16,17,18,19] and program dependency graph based approaches [20]. AST-based approaches in general are more accurate, and could report more clones than text-based and/or token-based approaches, but various techniques are needed to make them scalable. A comparison and evaluation of these techniques in terms of recall and precision as well as space and time requirements has been conducted by Bellon et. al., and the results are reported in [21].

Closely related work to ours is by Bulychev et al. [19] who also use the notion of anti-unification to perform clone detection in ASTs. Our approach is different from Bulychev et al.'s in several aspects. First, we use a different approach,

which is faster but reports more false positives, to get the initial clone candidates; second, their approach reports only clone pairs, while our approach reports clone classes as well as their anti-unifiers; third, Bulychev et al.'s approach is programming language independent, and the quality of the algorithm depends on whether the occurrence of the same variable (in the same scope) refers to one leaf in the AST; whereas our tool is for Erlang programs, though the idea also applies to other languages, and static semantics information is taken into account to disallow inconsistent substitutions.

In [22], Brown and Thompson describe an AST-based clone detection and elimination tool for Haskell programs. While their approach works on small Haskell programs, scalability is still the problem for the authors to address.

ClemanX [23] is an incremental tree-based clone detection tool developed by Nguyen et al. Their approach measures the similarity between code fragments based on the characteristic vectors of structural features, and solves the task of incrementally detecting similar code as an incremental distance-based clustering problem. In [24], Göde and Koschke describe a token-based incremental clone detection technique, which makes use of the technique of generalised suffix trees.

8 Conclusions and Future Work

We have presented an incremental clone detection and elimination technique which can be used interactively during a clone inspection and elimination process, but also of particular value both to the working programmer and the project manager of a larger programming project, in that it allows clone detection to contribute to the 'dashboard' reports from nightly builds, for instance.

For a tool to be used in an interactive way, performance and efficiency are especially important. This goal is achieved by our tool to incrementally update the clone report after changes have been made to the program. Being able to specify various parameter settings to the clone detector, the user has more control of the granularity of the clones reported. Using the AST as the internal representation of Erlang programs, and being static semantics aware, the clone detection tool achieves 100% accuracy, which is essential when clone elimination is concerned. To support clone elimination, our tool reports not only the common abstraction of a clone class, but also the application instance of each clone class member. The tool has been used in various industry case studies [10], during which its usability has been improved, and usefulness has been demonstrated.

As future work, we plan to extend the tool to detect expression sequences which are similar up to a single insertion or deletion of an expression, or similar up to a number of expression-level edits. Our overall approach is language-independent, and we also plan to apply our techniques to clone detection and elimination to test languages such as TTCN-3.

This research is supported by EU FP7 collaborative project ProTest (<http://www.protest-project.eu/>), grant number 215868.

References

1. Plotkin, G.D.: A Note on Inductive Generalization. *Machine Intelligence* 5 (1970)
2. Reynolds, J.C.: Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence* 5, 135–151 (1970)
3. Li, H., Thompson, S.: Similar Code Detection and Elimination for Erlang Programs. In: *Practical Aspects of Declarative languages* 2010, pp. 104–118 (2010)
4. Cesarini, F., Thompson, S.: *Erlang Programming*. O'Reilly Media, Inc., Sebastopol (2009)
5. Li, H., et al.: Refactoring with Wrangler, updated. In: *ACM SIGPLAN Erlang Workshop 2008*, Victoria, British Columbia, Canada (2008)
6. Baader, F., Siekmann, J.H.: Unification Theory. In: *Handbook of logic in artificial intelligence and logic programming*, pp. 41–125 (1994)
7. Ukkonen, E.: On-Line Construction of Suffix Trees. *Algorithmica* 14(3) (1995)
8. Baker, B.S.: On Finding Duplication and Near-Duplication in Large Software Systems. In: Wills, L., et al. (eds.) *WCSE*, Los Alamitos, California (1995)
9. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Computer Society Trans. Software Engineering* 28(7), 654–670 (2002)
10. Li, H., et al.: Improving your Test Code with Wrangler. Technical Report 4-09, School of Computing, University of Kent (2009)
11. Tiarks, R., Koschke, R., Falke, R.: An Assessment of Type-3 Clones as Detected by State-of-the-art Tools. In: *SCAM 2009*, Los Alamitos, CA, USA (2009)
12. Roy, C.K., et al.: Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Sci. Comput. Program.* 74(7) (2009)
13. Baker, B.S.: A Program for Identifying Duplicated Code. *Computing Science and Statistics* 24, 49–57 (1992)
14. Ducasse, S., Rieger, M., Demeyer, S.: A Language Independent Approach for Detecting Duplicated Code. In: *Proceedings ICSM 1999*, pp. 109–118. IEEE, Los Alamitos (1999)
15. Li, Z., Lu, S., Myagmar, S.: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. Softw. Eng.* 32(3), 176–192 (2006)
16. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone Detection Using Abstract Syntax Trees. In: *ICSM 1998*, Washington, DC, USA (1998)
17. Evans, W., Fraser, C., Ma, F.: Clone Detection via Structural Abstraction. In: *The 14th Working Conference on Reverse Engineering*, pp. 150–159 (2008)
18. Jiang, L., Mishergi, G., Su, Z., Glondu, S.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In: *ICSE 2007*, pp. 96–105 (2007)
19. Bulychev, P., Minea, M.: Duplicate Code Detection using Anti-unification. In: *Spring Young Researchers Colloquium on Software Engineering*, pp. 51–54 (2008)
20. Komondoor, R., Horwitz, S.: Tool Demonstration: Finding Duplicated Code Using Program Dependences. In: Sands, D. (ed.) *ESOP 2001*. LNCS, vol. 2028, p. 383. Springer, Heidelberg (2001)
21. Bellon, S., Koschke, R., Society, I.C., Antoniol, G., Krinke, J., Society, I.C., Merlo, E.: Comparison and Evaluation of Clone Detection Tools. *IEEE TSE* 33 (2007)
22. Brown, C., Thompson, S.: Clone Detection and Elimination for Haskell. In: *PEPM 2010: Partial Evaluation and Program Manipulation*, pp. 111–120 (2010)
23. Nguyen, T.T., Nguyen, H.A., Al-Kofahi, J.M., Pham, N.H., Nguyen, T.N.: Scalable and Incremental Clone Detection for Evolving Software. In: *ICSM 2009* (2009)
24. Göde, N., Koschke, R.: Incremental Clone Detection. In: *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering* (2009)

Analyzing Software Updates: Should You Build a Dynamic Updating Infrastructure?

Bashar Gharaibeh, Hridesh Rajan, and J. Morris Chang

Iowa State University

Abstract. The ability to adapt software systems to fix bugs, add/change features without restarting is becoming important for many domains including but not limited to finance, social networking, control systems, etc. Fortunately, many ideas have begun to emerge under the umbrella term “dynamic updating” to solve this problem. Dynamic updating is critical to address certain software evolution needs. Dynamic updating literature evaluates such systems in terms of coverage (i.e. what type of code changes are supported) and performance. However, we do not have a technique to analyze whether certain updating solution, based on its costs and benefits, is suitable for an application.

In this paper, we present a quantitative analysis model to fill this gap. Our model is parameterized and it can be instantiated with application-specific valuation functions. Given the software evolution history of the application under consideration, our model allows rigorous comparisons of the value of different software updating schemes (e.g. online vs. offline). We illustrate our model using two case studies inspired from the evolution history of Xerces XML parser library and Apache httpd web server. Other case studies and evaluation examples are presented in our technical report [Gharaibeh, Rajan and Chang 09]. The proposed analysis scheme can serve system architects in evaluating their current updating scheme. For example, to audit the system’s value during previous development cycles and whether a different updating scheme will generate higher value.

1 Introduction

Software evolution and maintenance is a fact of life [3, 15]. Enhancements, security, and bug fixes are routinely made to a software system during its usable lifetime. Long running software systems such as web and application servers, financial software, critical control systems often need to balance evolution and availability requirements. For such systems, downtime due to software update is unacceptable and often very costly [14, 17, 26].

Dynamic software updating has attracted significant interest in the last few years [6, 20, 24]. This is due to the benefits software updating can provide to long running applications. The interest in dynamic updating is clear from a plethora of research efforts and a specialized workshop (i.e. HotSwUp). Such interest is only expected to continue with the industrial trends toward software as long-running services in service-oriented architectures.

However, adopting any dynamic updating scheme requires deep understanding about its cost and benefits beyond the stated software engineering benefits. To date, dynamic

updating literature evaluates such systems in terms of coverage (i.e. what type of code changes are supported) and performance. For example, Chen *et al.* [6] evaluated their system over a set of server applications. The evaluation was in terms of average server's response time before and during the update process.

What is missing is a formal quantitative analysis that allows us to study such a system in comparison to current static update practices or other dynamic updating systems. We need to answer the question of whether the benefits of dynamic updating justifies its cost (performance or regular fees). In theory, being online 24/7 is a priceless advantage. However, this may not apply to all systems. Given the real history of bugs in a particular software, does the loss of system value due to these bugs justifies the investment in dynamic updating? The answer depends on many factors related to system operations and bugs severity.

The contribution of this work is a quantitative value model that allows us to study the gain from updating systems. Our model is based on Net option-value (NOV) analysis [29]. NOV has been devised to price options in a financial market and has also been used to study the cost and benefit of modularity in designs [2, 16, 28]. Our value model allows us to study the relation between updating system's operational parameters (e.g. cost and timing) and value provided to users. To the best of our knowledge, this is the first attempt to quantitatively formulate and evaluate the costs/benefits of offline and dynamic updating in software systems.

The proposed model can be used in different scenarios. For example, it can be used to audit the system's value during previous development cycles. By using information about added features and their revenue, developers can compare the current update practice and whether a different update strategy would provide higher value. It can also be used to quantitatively compare different dynamic updating schemes. Given the characteristics of two updating schemes such as types of supported updates and performance characteristics, the two schemes can be quantitatively compared using a set of benchmark features.

We have applied our model to two case studies: the evolution of the XML parser library Xerces [30], and 42 bug fixes for Apache httpd server obtained from Bugzilla (Section 3). Other case studies are presented in our technical report [10]. Using these case-studies, we studied the model's trends, relative values depending on the selected parameters, and assessed its precision. These studies also give us insights on how one would actually go about estimating the parameters that serve as the input to the model. We believe this to be a very useful aide to system developers and maintainers. To summarize, our contributions in this paper are:

- A quantitative model for cost/benefit analysis of updating systems and its formulation. The novelty of the model is in its application of net options value theory to the area of software updates.
- Case studies from software evolution of real-world applications that illustrate the use of our model. The main benefit of the case studies is that they give insights into selection of the model parameters.

The rest of this paper is organized as follows. In Section 2 we discuss our quantitative model . We describe our case studies in Section 3. Section 4 presents the related

work while Section 5 discuss various aspects and limitations of our evaluation model. Section 6 discusses directions for future investigations and concludes.

2 Quantifying Software Update

This section presents our analysis model. The main idea behind the analysis model is the computation of daily revenue of the system. By understanding how different updating policies affect the daily value, we can calculate the effect on total revenue made by these systems.

2.1 Update Models

We will evaluate the following updating models:

- Model 0: Offline update at release time.
- Model 1: Offline update at feature time.
- Model 2: Dynamic Updating.

The first model (Model 0) represents the base case where updates are performed when a new version is released. The update in this model is performed offline so the service is stopped until the system finishes the updating process. The disadvantage here is that severe bugs will not be addressed in a timely manner.

The costs and benefits of the last two update models are summarized in Figure 1. Model 1 presents the option for offline updating at feature availability time. In this model, updates are scheduled on the next system restart and applied when the system goes offline. Users are able to install these features instead of waiting for the next release date. Under this model, users will be required to restart their phones, which might cause users to delay applying the patch until a more suitable time.

Finally, in Model 2 the system is dynamically updated when new features are available even if availability occurs before the next release time. However, users might suffer from short-time performance loss during the update process.

In the last two models, we assume that the system is restarted when a new version is released. Models 1 and 2 allow developers to deploy features quickly rather than waiting for the new version release time. However, under these models, a restart is still required at each release.

Model	Revenue
Model 1	(+) time value of feature. (-)cost of updating, which depends on cost of disabling and restarting the service.
Model 2	(+) time value of feature. (-)cost of online updating, which depends on feature complexity. (-)cost of using a modified system that supports online updating.

Fig. 1. Value of Updating to Feature i

2.2 Net Options Value Model

Net Options Value (NOV) model quantifies the value of using the system over a certain period of time. In other words, if the value is represented as a function of time, the total value is equal to the integration of the value function over the specified period.

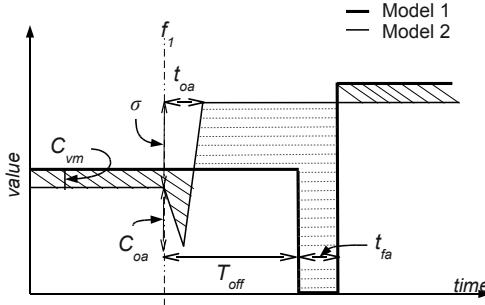


Fig. 2. Net Option Value of Different Updating Models

To illustrate, consider the scenario in Figure 2. It shows the revenue generated by Model 1 (bold line) and Model 2. Each model's total revenue is equal to the area under its value function. Model 2 has less value initially due to the cost of supporting dynamic updates. However, Model 2 gains value by early adoption of feature and reduced cost of updating. The dip in the Model 2 value represent the cost of the updating process. For Model 1, the dip is more severe since it incurs complete service disruption. The area in the figure shaded by diagonal lines represents the gain achieved by offline over dynamic updating, while areas shaded by horizontal lines represents the gain of dynamic over offline updating. Intuitively, if the area of diagonally shaded region is larger than the horizontal region, then offline updating provides better total revenue and the cost of supporting dynamic updating does not justify its benefits.

In general, net options value [2] is represented as follows:

$$V = S + \sum_i NOV_i - C$$

$$NOV_i = V_i - C_i$$

where S is the base system's value (i.e. before applying new features), V is the net value of the model, C is the model cost, which is paid even if no updates were exercised. NOV_i is the value gained by updating to feature i and C_i is the cost of the update. This formula, although general, does not offer much insight into the specifics of a typical updating system. Thus, we seek a domain-specific formulation of the net-options value analysis starting with a quantitative treatment of the value of Model 1 and 2.

Model 0: Static Update at Release Time. For this model the system value increases at release time by an amount equal to added features value. Thus we define the system value (V) for this model at a future release as:

$$V = S + \sum_i \sigma_i$$

where S is the system value at the current release and σ_i is the technical significance (value) of feature i . In other words, the value of the system after installing a new release is equal to its original value (old release value) plus the value of new features.

Model 1: Static Update at Feature Time. For this model the system value increases at next restart time by an amount proportional to added features time value. The cost has two components. First, the cost of delaying the update. Second, the cost of restarting the service. Thus we define the system value (V) for this model at a future release time (t_i) until the new version is released T , as follows:

$$V = \sum_{i=1}^n NOV_i$$

$$NOV_i = E[U] \int_{t_i + T_{off}^i}^T \sigma_i(t) dt - C_R \quad (1)$$

$$C_R = \begin{cases} 0 & t_i + T_{off}^i = t_{i-1} + T_{off}^{i-1} \\ U_L \int_0^{t_{fa}} dt \underbrace{\sum_{j=1}^{i-1} \sigma_j(t_j)}_{*} & \text{otherwise} \end{cases} \quad (2)$$

(3)

The value function we will use represent the value gained by a single user. It is often necessary to multiply the gained value by the expected number of users to obtain the total value. In the value model, $E[U]$ represents the expected number of users, U_L is the number of users at low-demand time. T_{off} is expected value of time until update is applied, and t_{fa} is the time needed to complete offline update. This value model has two parts. The first part describes how the deployment of a feature increases the system value. The value is equal to the summation of daily revenue of a feature represented by ($\sigma(t)$). The integration bounds represents the period of time the new feature is active. Since this model relies on scheduled restarts, the feature will not be deployed at its release time (t_i) but rather after certain number of days (T_{off}). The second part of the formula presents the cost associated with offline updating. The first case states that if two features are scheduled on the same restart period, we only need to pay the cost once. The second case presents the cost of the restart in terms of lost value (system value so far, labeled with (*)) and the time needed to finish the restart of the system after update (t_{fa}).

Model 2: Dynamic Updating. For this model the system value increases at feature availability time by an amount proportional to added feature's time value. The cost has two components. First, the long-running cost of using the updating system. Second, the cost of performing the update. Thus we define the system value (V) for this model at a future release as:

$$V = E[U]C_{vm} \sum_{i=1}^n NOV_i$$

where C_{vm} is the ratio of the performance of a dynamic-updating system relative to an offline-updating system and ranges over the period $[0, 1]$, where having the value of one means that there are no long-running overhead. Thus, as $C_{vm} \approx 1$, the operating cost of the system with dynamic updating decreases. Like Model 1, $E[U]$ is the expected number of users. The per-feature value (NOV_i) is defined as follows:

$$NOV_i = \int_{t_i}^T \sigma_i(t) dt - \int_0^{t_{oa}} C_{oa}(t) dt \sum_{j=1}^{i-1} \sigma_j(t_i) dt \quad (4)$$

where t_i is the time of release for feature i , T is the time of next release, $\sigma_i(t)$ the value function of the feature, t_{oa} is time needed to finish the dynamic update, and C_{oa} represents the reduction in system's value during the dynamic updating. Again, this value model represents the gain from deploying the feature (integration of $\sigma(t)$) minus the cost of the dynamic update which is related to update duration and value loss during the update.

2.3 Effect of Operational Parameters

Operational parameters are those used to describe the cost and timing of the update process. Based on the previous valuation models, we will now construct a set of relations that describes the bounds on these parameters that guarantees profitable operation. The original value models can be used to compare total revenue, while this set of relations can be used to calculate the system parameters based on known constraints.

Effect of Updating Overhead. In our model, both update systems suffer a value loss during the update. However, the dynamic update system also pays the continuous cost of supporting dynamic updates (C_{vm}). The value of C_{vm} represents the performance overhead from using the dynamic update system. It is known that such overhead must be kept at minimum. However, the question is when does the overhead reverse any gains from the modified system.

In general, the relation between C_{vm} and gain in comparison to the other system can be modeled by equating equations (4) and (3). By assuming n dispersed features, dynamic updating has higher value when its value is higher than the value offered by static updating. After simplification, the effect of update overhead is presented in the following formula:

$$\begin{aligned} & \overbrace{\sum_{i=1}^n [E[U](1 - C_{vm}) \int_{t_i}^T \sigma_i(t) dt + E[U]C_{vm} \int_0^{t_{oa}} C_{oa}(t) dt \sum_{j=1}^{i-1} \sigma_j(t_i) dt]}^{\text{effect of dynamic update}} \\ & - E[U] \underbrace{\int_0^{T_{off}^i} \sigma_i(t) dt - U_L \int_0^{t_{fa}} dt \sum_{j=1}^{i-1} \sigma_j(t_i) dt}_{\text{effect of static update}} < 0 \end{aligned}$$

The first half represents the cost of the dynamic update system which consists of the long-running cost and the update cost. The second half shows the offline updating cost

consisting of delayed feature deployment and service disruption at update time. Notice that as C_{vm} increases to reach the value of one (no long-running costs), the cost of dynamic update is reduced to the cost of the update process at update time. As C_{vm} decreases, the long running cost increases in a similar amount. Also, note that as either T_{off} or t_{fa} increases, lower values of C_{vm} can be tolerated.

Effect of Delayed Updates. For two features f_i, f_j where $t_j > t_i$, applying the two features at t_j has higher value than applying each feature at its time for Model 1 if (here terms have their previously defined meanings):

$$U_L \int_0^{t_{fa}} dt \sum_{k=1}^{i-1} \sigma_k(t_i) > E[U] \int_{t_i + T_{off}^i}^{t_j + T_{off}^j} \sigma_i(t) dt$$

and for Model 2 if

$$\int_0^{t_{oa}} C_{oa}(t) dt \sum_{k=1}^{i-1} \sigma_k(t_i) > \int_{t_i}^{t_j} \sigma_i(t) dt$$

On the other hand, if $\sigma(t), C_{oa}(t)$ do not depend on time (i.e. constant values), these conditions are simplified to:

$$\begin{aligned} \text{Model 1: } \sigma_i &< \frac{U_L}{E[U]} \frac{t_{fa} \sum_{k=1}^{i-1} \sigma_k(t_i)}{(t_j + T_{off}^j) - (t_i + T_{off}^i)} \\ \text{Model 2: } \sigma_i &< \frac{t_{oa} C_{oa} \sum_{k=1}^{i-1} \sigma_k(t_i)}{t_j - t_i} \end{aligned} \quad (5)$$

The later condition relates the value of σ_i to the cumulative system value, update cost and period between features. For example, under Model 2 (5), a feature that is equal to 10% of cumulative value and with a period of one week until next feature, the dynamic update time should be more than 8 hours and 24 min to justify combining the update of these two features.

Coverage of Dynamic Updating. Many dynamic updating systems do not support all types of code updates. Therefore, even a dynamic update system requires occasional restarts to serve certain update requests. Generally, we can include this factor as a random event x_i that is related to the ratio of supported updates. Assuming that for any certain feature, there is a probability $p(x_i)$ that the feature can not be updated dynamically. Therefore, the valuation model of Model 2 is changed as follows:

$$\begin{aligned} NOV_i = & p(x_i) NOV_i^1 \\ & + (1 - p(x_i|x_{i-1})) \max\{NOV_i^1, NOV_i^2\} \\ & + (1 - p(x_i|\bar{x}_{i-1})) NOV_i^2 \end{aligned}$$

In the new model, the NOV of feature i has two factors. First, there is a probability of x_i that a static update is required (i.e. NOV_i^1). Second, if the feature can be applied dynamically, the NOV is the maximum of the dynamic and static NOV. We are using the maximum aggregate to cover the possibility that feature $i - 1$ was updated statically

(i.e. $p(x_i|x_{i-1})$) and that the new feature is released within the T_{off} period. In this case, we have the option of upgrading feature i dynamically at the regular cost or statically at reduced cost since the restart is already required. Otherwise, the regular NOV of dynamic update is used (i.e. $p(x_i|\bar{x}_{i-1})$)

3 Applying Our Analysis Model

This section applies our analysis model for comparing software updating schemes to Apache httpd [1], a well-known web server. The main objective is to study our model's trends, relative values depending on the selected parameters, and assess its precision. The Apache httpd case we collected information about bug fixes over a five years period.

We will start by describing the process of selecting the evaluation parameters. Then we will present detailed information about the case study. Finally, we will study the effect of operating parameters on gains achieved by different updating models and how the timing of applying updates affect the system's value.

3.1 Selecting Analysis Parameters

The main challenge in the application of our model is selection of proper value functions. Each feature contributes to the value of a release and each bug reduces the system value until it is fixed. However, assigning proper values of $\sigma(t)$ is not trivial as it requires an understanding of the technical importance of a feature and how it affects the whole system's value. Here, we use a simple heuristic to evaluate a feature's importance. For evaluation purposes, we used a constant value for $\sigma(t)$. However, if more information is available regarding a feature effect on system's value, this information should be represented using a more appropriate function. The value of T_{off} equals the number of days until offline updates are performed (i.e. Sundays). The value of t_{oa} is approximated by αt_{fa} . The value of α depending on code modifications required to implement the feature and was computed by studying code changes. Finally, the value of C_{oa} is set to 0.5. This value indicates that the system loses half of its performance (which is very conservative) during the dynamic update process.

3.2 Xerces Case Study

We selected ten features from two consecutive releases of Xerces XML parsing library. The features provide additional capabilities (e.g. A3: Japanese characters serialization, B4: support for <redefine> attribute), performance enhancement (e.g. A4: improve Deterministic Finite Automaton(DFA) build-time performance) or resolve bugs (e.g. B1). Deploying these features allows the system to increase its revenue through faster processing, wider customer base and support additional types of XML documents. We approximated σ_i values for studied features through a point system. We assume that a system value ($\sum_i \sigma_i$) doubles at every release. Any security-related features is assigned four points, bug fixes and added features are assigned three points, performance enhancement are assigned two points, and finally, any remaining features are assigned one point. Using this approach, each feature's σ_i is equal to its share of points.

Feature	Points	σ	α	$T - t$	T_{off}
A1	3	0.214	0.185	54	2
A2	3	0.214	0.012	50	5
A3	3	0.214	0.235	48	3
A4	2	0.143	0.136	20	3
A5	3	0.214	0.432	13	3
B1	3	0.214	0.4	43	3
B2	3	0.214	0.36	42	2
B3	2	0.143	0.1	38	5
B4	3	0.214	0.08	28	2
B5	3	0.214	0.05	11	6

Fig. 3. Xerces Feature's Parameters

For simplicity, we are assuming that a release consists of these features only. Figure 3 shows the parameter values for selected features. The table shows the number of points assigned to each feature as points are used to approximate value gained by deploying a feature (σ). The table also shows the feature's relative complicity (α) as derived from code modification logs. Feature complexity is used to derive the dynamic updating time (t_{oa}). A complex feature requires more updating time than a simpler feature. The table also lists the time in days until the next release is available ($T - t$) and the wait period from the feature release time until the next Sunday (T_{off}) which we used as waiting period for the offline updating.

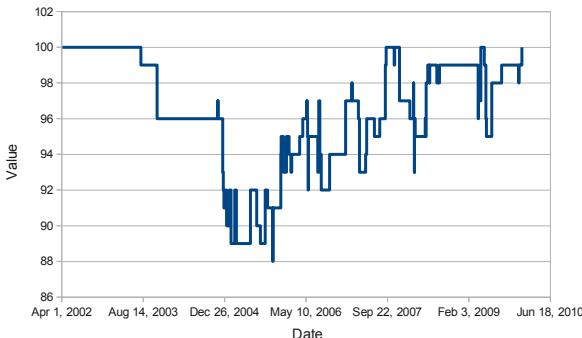
We can note that feature complexity follows the trend of feature's value for Xerces. In other words, important features are complex. Therefore, supporting a high-value feature comes at higher cost than a simpler feature, but will provide higher value.

3.3 Apache httpd Case Study

In this case study, we analyzed the history of bug fixes for Apache httpd for versions from 2.0 to 2.3. Figure 4 shows an overview of bugs timeline and their effect on system value. Each bug has a severity level that represents its effect on the system and is assigned by users reporting the bug. In the figure, the system value is decreased by the weight of the bug severity (Figure 5) starting from bug discovery date until it was fixed. In this study, bugs that can be fixed by changing configuration files rather than source code, and bugs that caused a crash at startup were excluded. These bugs require a restart in any update model and thus, will not be included in the analysis between dynamic and static updating. Furthermore, we excluded bugs with severity below normal. The reason behind excluding these bugs is that users do not usually update their servers for below normal bugs.

The value of each bug fix (W) is proportional to its severity. Since severity is selected by users reporting the bug, it reflects the value of the bug fix more accurately compared to value assigned by an external observer. However, it is less obvious how bug severity affects the system quantitatively. For httpd, we assigned different weights to different severity levels (Figure 5)¹. To compute σ_i , the severity weight is multiplied by β , which is an estimate of value loss. For example, a packet monitoring service may

¹ <https://issues.apache.org/bugzilla/page.cgi?id=fields.html>

**Fig. 4.** History of httpd Value

Category	W	Description
Critical	3	crashes, loss of data, memory leak
Major	2	major loss of function
Normal	1	some loss of functionality under specific circumstances

Fig. 5. Apache httpd Bug Categories. Taken from ASF Bugzilla: A Bug's Life Cycle.

be employed to record transactions affected by the bug. Such system can affect the overall throughput, and thus, reduces the system value (i.e. hits per day).

The value of a bug fix depends on its severity level (user supplied) and its estimated value loss (β). We later show the effect of β and bug severity on system values of different updating models.

3.4 Analysis

We will now apply our analysis model to evaluate the two update models (Model 1 and Model 2). We will study the difference and the effect of operational parameters on revenue.

Revenue Analysis. Assuming $E[U] = U_L = 1$, Figure 6 presents the revenue values for Model 2 and Model 1. These values reflect the expected benefits for a single continues user. Note that increasing the number of expected users ($E[U]$) will increase the absolute revenue. However, it has minimum effect on the difference between Model 1 and Model 2 update systems. The main cause of increased value in Model 2 for Xerces is the wait period until restart required by Model 1. For Apache httpd, the wait period for receiving a bug fix is manifolds longer than that for the next scheduled restart. Therefore, in the case of httpd, the long running cost of Model 2 makes it less beneficial on the long run.

Effect of Updating Overhead. Figure 7 shows the gain percentage from using Model 2 compared to Model 1 for the studied features from Xerces and bug fixes of Apache httpd. It shows that for Xerces, dynamic updating can provide benefit as long as its

Cycle	Model 1	Model 2
Xerces 1.2.3-1.3.0	34.86	37.73
Xerces 1.3.0-1.3.1	28.43	31.64
httpd	2285.15	2269.66

Fig. 6. NOV Calculation when $E[U] = U_L = 1$ and $t_{fa} = \text{one min}$. $C_{vm} = 0.99$. $\beta = 0.05$.

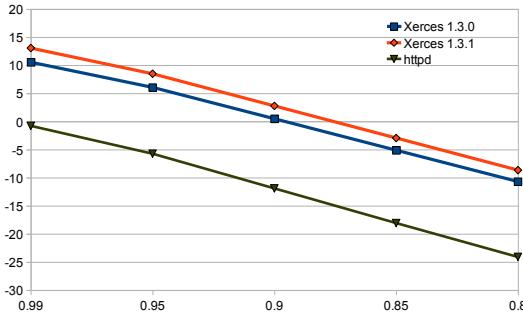


Fig. 7. Effect of C_{vm} . $E[U] = U_L = 1$ and $t_{fa} = \text{one min}$

performance is above 90% of Model 1 performance. In other words, the long running costs of using dynamic updating for Xerces must not exceed 10% of the system revenue. If supporting the dynamic update system reduces a server's performance (e.g., satisfied requests per second) by 10%, then this performance loss translates into lost customers, and thus a loss in revenue by 10%. Any gain from early adoption of features will be eliminated by the constant high cost of supporting dynamic updating. Apache httpd presents a different story. Even at highest C_{vm} ratio, there is no benefit from using Model 2. While as expected, the loss of value increases as C_{vm} decreases.

Effect of Restart Schedule. The main reason that Model 2 can generate better value compared to Model 1 is delayed updates in Model 1, which is related to T_{off} . This parameter represents the period of maintenance cycle in model 1. High value indicates longer periods without restarts and thus reduced cost due to service interruption. On the other hand, low values of T_{off} brings required updates at a faster rate. Figure 8 shows the relation between the value of T_{off} and the gain of model 2 compared to Model 1. At higher T_{off} values, the static update model losses most of its benefits and become closer to Model 0. Xerces case is very sensitive to varying the value of T_{off} due to the short period between features. As T_{off} increases, many features will be delayed. However, Apache httpd bug fixes are well-dispersed in time. Therefore, higher T_{off} barely affect Model 2 gain (from -1% to 1.5%).

With low T_{off} (i.e. daily restarts), the system will closely follow the value of Model 2. It is worthy to note that in all cases, we assumed that static updates occur on low demand times (i.e. restart cost multiplied by U_L rather than $E[U]$). In reality, this assumption may not hold for low values of T_{off} .

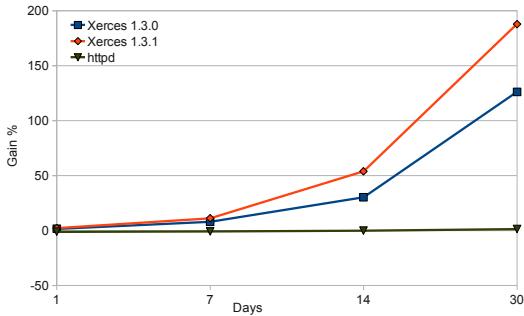


Fig. 8. Effect of t_{off} . $E[U] = U_L = 1$ and $t_{fa} = \text{one min}$. $C_{vm} = 0.99$.

3.5 Summary

In this section, we investigated the value of different update models on a set of real-world applications (Xerces and Apache httpd). Our main objective was to study our model's trends and values depending on the selected parameters. Several key insights are worthy to note. First, the long running cost of dynamic updating has an influential role in determining the total revenue. Another observation is the relation between bug fix history and benefits from dynamic updating. In general, we note the usefulness of this model in understanding why certain models perform better and how the model parameter affects its performance. We also note that the exact values obtained in our analysis are dependent on the choice of the model parameters values.

4 Related Work

Dynamic updating is gaining increased interest from research and industry. Several research projects have proposed, designed and implemented dynamic updating systems. However, the main evaluation tasks in the literature were performance and coverage. Chen *et al.* [6] and Subramanian *et al.* [27] evaluated their systems in terms of service disruptions during the update process. Evaluation of runtime aspect-weaving tools [8] have also focused on runtime overhead. Dumitras *et al.* [7] developed a framework to assess the risk of dynamic update verses the risk of postponing the update. In this paper, we explored a different evaluation goal and methods. To the best of our knowledge, this is the first exposition into evaluating update systems in terms of running costs and added options value.

Our evaluation model is based on the NOV analysis [5, 13]. NOV analysis is based on the problem of pricing financial options. A financial option presents the opportunity to purchase a commodity at a strike price in the future regardless of price fluctuations, provided that the buyer pays a premium in the present (also known as Call Option). In this paper we used the basics of options analysis to evaluate the benefits of dynamic updating. Updating has a significant resemblance with the problem of option pricing. As options, dynamic updating provides the opportunity to perform a future update at a

possibly reduced price given that a premium (i.e. cost of using the dynamic update system) is paid. The body of literature describing this financial instruments is extensive and out of the scope of this paper. However, we note the application of options to software design and especially to design modularity. Baldwin and Clark [2] showed the benefits of modular design in increasing a system's value. They conclude that a set of options over modules are more valuable than options on the whole system. This idea is further utilized in software design research by analyzing which modularization provides the best value. Sullivan *et al.* [28] show the value of design based on information hiding principles by combining NOV analysis and design information.

Similar uses of option analysis can be found in [4, 12, 16]. Our work shares the basic analysis techniques since the problem of quantifying updating benefits can be translated into a modular design evaluation problem (i.e. Updatable systems are modular). However, the case for software updating presents a different set of operational parameters and dependencies on time that are not considered for option analysis for software design. Ji *et al.* [11] used option analysis to evaluate the benefits from designing and issuing new software releases in relation to market uncertainty. Their analysis is concerned with the software developer perspective and analyze the preferred market conditions for releasing an upgrade (additional features). In this paper, we were mainly concerned with how to decide between different upgrading policies. In contrast, our analysis assists system users and updatable systems designers, rather than feature providers, with deriving decisions related to upgrading policies.

The problem of designing dynamically updatable systems has also received considerable attention in the last decade. Oreizy *et al.* [22, 23, 25] and Garlan *et al.* [9] have presented and studied dynamic software architectures. These systems were evaluated based on the performance of resulting application and other code metrics. The model presented in this paper can be applied to evaluate different online updating schemes including those presented by Oreizy *et al.* and Garlan *et al.*. Evaluating dynamic deployment architectures were also presented Mikic-Rakic in her PhD thesis [18], where the goal was to reduce service disruption (i.e. increase value) in distributed systems through better deployment strategies. In this paper, we are not concerned with enhancing a specific updating system, but rather on providing a mechanism to evaluate and compare their benefits.

5 Discussion

We have illustrated how our proposed model can be used to evaluate updating systems and to understand the effect of some operational parameters. This evaluation model is advantageous since it accounts for the value of time and supports the study of time dependent value functions. In our evaluation (Section 3), we treated the feature value function as a constant. In general, assigning values to features is often subjective. However, it would be of interest to study value functions that directly depend on time. For example, functions that model compound interest on feature's value.

We assumed that each applied update is correct and does not fail (i.e. bug-free). This assumption simplifies the formulation. However, a more practical model will incorporate the possibility of failed updates. A failed update can be considered as a feature

with negative gain to model value loss during the use of the malfunctioning code. Since failures are unknown before their occurrence, this additional negative-gain feature will depend on a probability distribution that describes bug probability over time. The issue of failed updates have been extensively studied by Mokous and Weiss [19].

Finally, this study evaluated two update models, static(offline) and dynamic. An interesting question is to try to evaluate a combination of several dynamic update schemes depending on the nature of the feature and how they compare in provided value.

6 Conclusions and Future Work

Software updating has several advantages such as runtime monitoring, bug fixes or adding features to long running applications. Therefore, dynamic software updating has attracted significant interest in the last few years [6, 20, 21, 24, 26]. To date, dynamic updating literature evaluates such systems in terms of coverage (i.e. what type of code changes are supported) and performance. For example, Chen *et al.* [6] evaluated their system in terms of service disruptions during the update process and noted the types of code changes that their system can not handle. Such evaluation is sufficient to understand the system performance and coverage. However, we often need other metrics to compare different updating systems. For example, what would be the gain from dynamic updating over offline updating, or what is the gain difference between two dynamic updating systems. To answer these questions, we formalized a quantitative model to evaluate the net revenue gained by the use of different updating models. Using this model, we were able to evaluate the gain from online updating vs. offline updating based on the evolution history of real-world applications. Furthermore, the model can also be used to compare two, updating schemes that differ in their coverage and performance.

An interesting outcome of this analysis was an insight into the perceived value of performance overheads for dynamic update systems. Generally, researchers have been concerned about two kinds of such overheads [8]: first, during update time, and second, constant overhead during the system's normal execution. Our analysis provides a method to analyze and compare these overheads based on their perceived values, which has the potential to aid in the selection of an updating system during software design.

Future work involves extending our analysis model in two main directions. First, the formulation can be extended to model the effect of bug discovery. Often after a feature release a bug is discovered and a second patch is needed to resolve the bug. The extension can model the revenue loss from such activity. Second, in terms of evaluation, we used simple constants to represent feature values. However, modeling real-world economics would require more complex valuation functions.

References

1. Apache, <http://httpd.apache.org/>
2. Baldwin, C.Y., Clark, K.B.: Design Rules: The Power of Modularity, vol. 1. MIT Press, Cambridge (1999)
3. Bennett, K.H., Rajlich, V.T.: Software maintenance and evolution: a roadmap. In: The Conference on The Future of Software Engineering, pp. 73–87 (2000)

4. Cai, Y.: Modularity in Design: Formal Modeling and Automated Analysis. PhD thesis, U. of Virginia (2006)
5. Chandler, A.D.: Strategy and Structure. MIT Press, Cambridge (1962)
6. Chen, H., Yu, J., Chen, R., Zang, B., Yew, P.-C.: POLUS: A POrter Live Updating System. In: ICSE (2007)
7. Dumitras, T., Narasimhan, P., Tilevich, E.: To upgrade or not to upgrade: impact of online upgrades across multiple administrative domains. In: OOPSLA 2010, pp. 865–876 (2010)
8. Dyer, R., Rajan, H.: Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In: AOSD 2008 (2008)
9. Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37, 46–54 (2004)
10. Gharaibeh, B., Rajan, H., Chang, J.M.: A quantitative cost/benefit analysis for dynamic updating. Technical report, Iowa State University (2009)
11. Ji, Y., Mookerjee, V., Radhakrishnan, S.: Real options and software upgrades: An economic analysis. In: International Conf. on Information Systems (ICIS), pp. 697–704 (2002)
12. Sullivan, K., et al.: Modular aspect-oriented design with XPIs. ACM TOSEM (2009)
13. Klepper, S.: Entry, exit, growth and innovation over the product life cycle. American Economic Review 86(30), 562–583 (1996)
14. Kniesel, G.: Type-safe delegation for dynamic component adaptation. In: Demeyer, S., Dannenbring, R.B. (eds.) ECOOP 1998 Workshops. LNCS, vol. 1543, pp. 136–137. Springer, Heidelberg (1998)
15. Lehman, M.: Software's future: managing evolution. IEEE Software 15(1), 40–44 (1998)
16. Lopes, C.V., Bajracharya, S.K.: An analysis of modularity in aspect oriented design. In: AOSD 2005, pp. 15–26 (2005)
17. Mätzel, K., Schnorf, P.: Dynamic component adaptation. Technical Report 97-6-1, Union Bank of Switzerland (1997)
18. Mikic-Rakic, M.: Software architectural support for disconnected operation in distributed environments. PhD thesis, University of Southern California (2004)
19. Mockus, A., Weiss, D.M.: Predicting risk of software changes. Bell Labs Technical Journal 5, 169–180 (2000)
20. Neamtiu, I., Hicks, M.: Safe and timely updates to multi-threaded programs. SIGPLAN Not. 44(6), 13–24 (2009)
21. Neamtiu, I., Hicks, M., Stoyle, G., Oriol, M.: Practical dynamic software updating for C. In: PLDI (2006)
22. Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime software adaptation: framework, approaches, and styles. In: ICSE Companion 2008: Companion of the 30th International Conference on Software Engineering, pp. 899–910 (2008)
23. Oreizy, P., Taylor, R.: On the role of software architectures in runtime system reconfiguration. In: Intl. Conf. on Configurable Distributed Systems, (1998)
24. Orso, A., Rao, A., Harrold, M.: A technique for dynamic updating of Java software (2002)
25. Oreizy, P., et al.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems 14(3), 54–62 (1999)
26. Malabarba, S., et al.: Runtime support for type-safe dynamic java classes. In: Hwang, J., et al. (eds.) ECOOP 2000. LNCS, vol. 1850, pp. 337–361. Springer, Heidelberg (2000)
27. Subramanian, S., Hicks, M., McKinley, K.S.: Dynamic software updates: a VM-centric approach. In: PLDI, pp. 1–12 (2009)
28. Sullivan, K.J., Griswold, W.G., Cai, Y., Hallen, B.: The structure and value of modularity in software design. In: ESEC/FSE 2001, pp. 99–108 (2001)
29. Williamson, O.E.: The Economic Institutions of Capitalism. Free Press, New York (1985)
30. Xerces. XML library, <http://xerces.apache.org/xerces-j/>

Flow-Augmented Call Graph: A New Foundation for Taming API Complexity

Qirun Zhang, Wujie Zheng, and Michael R. Lyu

Computer Science and Engineering
The Chinese University of Hong Kong, China
`{qrzhang, wj zheng, lyu}@cse.cuhk.edu.hk`

Abstract. Software systems often undergo significant changes in their life cycle, exposing increasingly complex API to their developers. Without methodical guidances, it is easy to become bogged down in a morass of complex API even for the professional software developers. This paper presents the Flow-Augmented Call Graph (FACG) for taming API complexity. Augmenting the call graph with control flow analysis brings us a new insight to capture the significance of the caller-callee linkages in the call graph. We apply the proposed FACG in API recommendation and compare our approach with the state-of-the-art approaches in the same domain. The evaluation result indicates that our approach is more effective in retrieving the relevant APIs with regard to the original API documentation.

Keywords: API Recommendation, Static Analysis, Control Flow Analysis.

1 Introduction

Software systems often undergo significant changes during the in-service phrase of their life cycle [7]. Most of contemporary software systems are becoming larger with exposing increasingly complex API to developers [8]. A recent survey conducted in Microsoft Research reveals that software developers usually become lost working on projects with complex API, unsure of how to make progress by selecting the proper API for a certain task [15]. Previous literature [6,20] points out that working with complex APIs in large scale software systems presents many barriers: understanding how the APIs are structured, selecting the appropriate APIs, figuring out how to use the selected APIs and coordinating the use of different APIs together all pose significant difficulties. Facing with these difficulties along, developers spend an enormous amount of time navigating the complex API landscape at the expense of other value-producing tasks [16].

A methodical investigation of the API usage in large software systems is more effective than an opportunistic approach [2]. The API relevance is usually considered to tame the API complexity. Two APIs are relevant if they are often used together or they share the similar functionality. According to previous literature [19], the original API documentation which groups the APIs into modules is the best resource to indicate the API relevance. However, few projects provide the insight to capture the relevant APIs in their documentation [8]. With the need exposed, recommendation systems specific to software engineering are emerging to assist developers [13]. Recommending relevant

APIs (namely, API recommendation) in software systems is also a long-standing problem that has attracted a lot of attention [4,8,9,14,19,22]. Generally, there are two fundamental approaches in API recommendation concerned with data mining techniques and the other with structural dependency. The mining approaches emphasize on “How API is used” and extract frequent usage patterns from *client code*. The structural approaches, on the other hand, focus on “How API is implemented” and recommend relevant APIs according to structural dependencies in *library code*.

Previous work on API recommendation heavily relies on the call graph. The call graph is a fundamental data representation for software systems, which provides the first-hand evidence of interprocedural communications [17]. Especially for the work concerned with API usage, the impact of the call graph is critical. The call graph itself does not imply the significance of callees to the same caller. However, the callees are commonly invoked by a caller under various constraints. For example, the callees enclosed with no conditional statement will be definitely invoked by the caller, whereas the callees preceded by one or more conditional statements are not necessarily invoked. Previous approaches relying on the call graph do not distinguish the differences between callees.

Our approach takes a different path from the previous work by extending the very foundation in API recommendation—the call graph. We introduce the Flow-Augmented Call Graph (FACG) that assigns weights to each callee with respect to the control flow analysis of the caller. The key insight of the FACG is that a caller is more likely to invoke a callee preceded by less conditional statements. Thus the bond between them is stronger comparing to the others preceded by more conditional statements. Therefore, the significance of caller-callee linkages can be inferred in the FACG. The insignificant callees in the FACG can be eliminated, so that the API complexity can be reasonably reduced with respect to specific software tasks. In this work, we employ the FACG in API recommendation and conduct our evaluation on several well-known software systems with three state-of-the-art API recommendation tools.

This paper makes the following contributions:

- We propose the Flow-Augmented Call Graph (FACG) as a new foundation for taming API complexity. The FACG extends the call graph by presenting the significance of caller-callee linkages in the call graph.
- We apply the FACG in API recommendation and evaluate our approach on several well-documented software projects. We employ their module documentation as a yardstick to judge the correctness of the recommendation. The evaluation indicates our approach is more effective than the state-of-the-art API recommendation tools in retrieving the relevant APIs.
- We implement our API recommendation approach as a scalable tool built on GCC. Our tool copes with C projects compliable with GCC-4.3. All the supplemental resources are available online¹.

The rest of the paper is organized as follows. Section 2 describes the motivating example. Section 3 presents the approach to build FACG and recommend relevant APIs.

¹ <http://www.cse.cuhk.edu.hk/~qrzhang/facg.html>

```

1 APR_DECLARE(apr_status_t) apr_pool_create_ex(...)

2 {
3     ...
4     if ((node = allocator_alloc(allocator, MIN_ALLOC - APR_MEMNODE_T_SIZE)) == NULL)
5     {
6         ...
7     }
8     ...
9 #ifdef NETWARE
10    pool->owner.proc = (apr_os_proc_t)getnlmhandle();
11 #endif /* defined(NETWARE) */
12    ...
13    if ((pool->parent = parent) != NULL)
14    {
15 #if APR_HAS_THREADS
16        ...
17        if ((mutex = apr_allocator_mutex_get(parent->allocator)) != NULL)
18            apr_thread_mutex_lock(mutex);
19 #endif /* APR_HAS_THREADS */
20        ...
21 #if APR_HAS_THREADS
22        if (mutex)
23            apr_thread_mutex_unlock(mutex);
24 #endif /* APR_HAS_THREADS */
25    }
26    ...
27    return APR_SUCCESS;
28 }

```

Fig. 1. `apr_pool_create_ex()` in Apache

Section 4 compares our approach with three state-of-the-art tools. Section 5 summarizes the previous work. Section 6 conducts the conclusion.

2 Motivating Example

We motivate our approach by selecting a real world API `apr_pool_create_ex()`² from the latest Apache HTTP server-2.2.16. Consider the code snippet shown in Fig. 1, if we investigate the control flow of this API, we may obtain two interesting observations. First, the call-site of API `getnlmhandle()` at line 10 is subject to the macro `NETWARE`³, and this API will never be called by `apr_pool_create_ex()` on the platforms other than Netware®. Second, the call-site of `allocator_alloc()` at line 4 is unconditional, whereas `apr_thread_mutex_lock()` at line 18 is subject to two conditions. As a result, `allocator_alloc()` is much more likely to be called by `apr_pool_create_ex()` than `apr_thread_mutex_lock()`. However, in the conventional call graph shown in Fig. 2(a), the callees are identical to the caller. These observations reveal that the conventional call graph is blind to the significance among different callees, and we are likely to miss some critical information if we treat every callee as the same in a call graph.

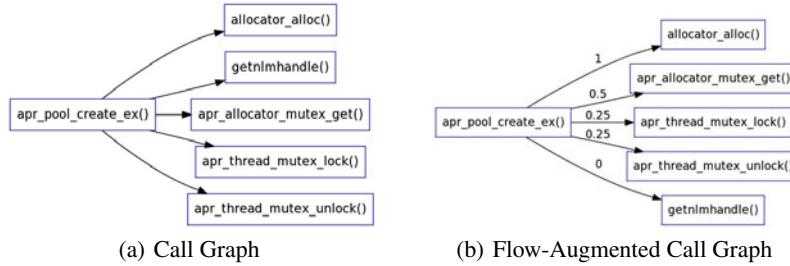
In this paper, we propose the Flow-Augmented Call Graph (FACG), which aims to address the limitations of the conventional call graph. We build the FACG with respect

² The irrelevant code is omitted.

³ Netware is a network operating system developed by Novell, Inc. Find more on <http://httpd.apache.org/docs/2.0/platform/netware.html>

Table 1. Flow distributions of each API in the previous example

API name	Description	Flow
apr_pool_create_ex	Creating a new memory pool.	
allocator_alloc	Allocate a block of mem from the allocator	1
apr_allocator_mutex_get	Get the mutex currently set for the allocator	0.5
apr_thread_mutex_lock	Acquire the lock for the given mutex.	0.25
apr_thread_mutex_unlock	Release the lock for the given mutex.	0.25
getnlmhandle	Returns the handle of the NLM owning the calling thread.	0

**Fig. 2.** Call Graph and Flow-Augmented Call Graph

to the fact that some of the call-sites are unconditional while others are conditional; some are concerned with more conditions while the others are with fewer conditions. By applying the control flow analysis, we observe that the unconditional call-sites occupy the caller's major control flow, whereas the call-sites under more conditions tend to reside in a less important sub branch. It is more possible for the caller to invoke the callee with fewer or without conditions. In order to cope with individual call-sites, we split the control flow equally for every branch in the control flow graph. For the motivating example in Fig. 1, we initialize the inflow by 1. The final distribution of the control flow for each callee is shown in Table 1. Especially, `getnlmhandle()` is eliminated since our analyzer is built on GCC Gimple IR, where all of the macros have been preprocessed by the compiler. Finally, we extract the description of each API from Apache Http Documentation⁴ to interpret the insight beyond our FACG. As shown in Table 1, among the five callees, API `allocator_alloc()` is more likely to accomplish "creating a new memory pool" than other less important APIs (e.g., `apr_thread_mutex_lock()`) in this case. The FACG shown in Fig. 2(b) indicates that the linkage of `apr_pool_create_ex()` and `allocator_alloc()` is the most significant one among all potential caller-callee pairs.

3 Approach

3.1 Augmenting the Call Graph with Control Flow

Parsing Source Code. The GCC compiler is chosen as the backbone parser. Our static analyzer takes the advantage of Gimple [10] Intermediate Representation and is able to

⁴ <http://apr.apache.org/docs/apr/1.4/modules.html>

capture essential information (e.g., basic block, API call-site and structure accessing) from the source code. The current implementation only works for C; however, it can be easily extended to other languages through different GCC front ends.

Reducing CFG. We adopt the definition of CFG from those presented by Podgurski and Clarke [11].

Definition 1. A Control Flow Graph (*CFG*) $G = (N, E)$ for procedure P is a directed graph in which N is a set of nodes that represent basic blocks in procedure P . N contains two distinguished nodes, n_e and n_x , representing ENTRY and EXIT node, where n_e has no predecessors and n_x has no successors. The set of N is partitioned into two subsets, N^S and N^P , where N^S are statement nodes with each $n_s \in N^S$ having exactly one successor, where N^P are predicate nodes representing predicate statements with each $n_p \in N^P$ has two successors⁵. E is a set of directed edges with each $e_{i,j}$ representing the control flow from n_i to n_j in procedure P . All nodes in N are reachable from ENTRY node n_e .

In the common case that the CFG is reducible [21], eliminating loop back-edges results in a DAG and this can be done in linear time[1]. For the irreducible CFG, we adapt the conservative approximation from [12] and unroll every loop exactly once. This is done at early stage so that the DAG instead of CFG is considered in building the FACG.

Calculating the Flow of Callees. For a callee Q , if a path with flow x contains Q , we say Q has flow x along the path, otherwise Q has flow 0 along the path. The flow of Q in the caller is the sum of the flow of Q along all the paths. To calculate the flow of Q , a naive strategy is to employ an exhausted graph walking strategy to collect the flow of callees in each path. However, this is infeasible as there are an exponential number of paths [12]. We propose an approach to calculate the flow of Q incrementally. We define the inflow and outflow of each basic block n_i as the following:

Definition 2. The inflow of a basic block is defined as:

$$IN(n_i) = \begin{cases} \sum_{e_{j,i} \in E} OUT(n_j) & if(n_i \neq n_e) \\ n_0 & if(n_i = n_e) \end{cases} \quad (1)$$

Definition 3. The outflow of a basic block is defined as:

$$OUT(n_i) = \begin{cases} IN(n_i) & if(n_i \in N^S) \\ \frac{IN(n_i)}{2} & if(n_i \in N^P) \end{cases} \quad (2)$$

The inflow $IN(n_i)$ denotes the flow of all the paths arriving the basic block n_i (n_i is counted in the paths), and the outflow $OUT(n_i)$ denotes the flow of all the paths

⁵ As indicated in [11], the outedge of each n_i in CFG is at most two. This restriction is made for simplicity only.

Algorithm 1. Algorithm to determine the flow of each callee in the DAG

Input : $G(V, E)$, directed, acyclic CFG of procedure P ;
 V is topologically sorted;
the set of Q , where Q is the callee of P ;

Output: $Q.Flow = IN(n_x)_Q$ for each callee Q ;

```

foreach callee  $Q$  of procedure  $P$  do
    foreach  $n_i \in V$  do
        foreach  $n_j \in PRED(n_i)$  do
             $IN(n_i) \leftarrow IN(n_i) + OUT(n_j);$ 
             $IN(n_i)_Q \leftarrow IN(n_i)_Q + OUT(n_j)_Q;$ 
        end
        if  $n_i$  has call-site of  $Q$  then
             $IN(n_i)_Q \leftarrow IN(n_i);$ 
        end
    end
     $Q.Flow \leftarrow IN(n_x)_Q$ 
end
```

arriving a successor of n_i through n_i . We also use $IN(n_i)_Q$ and $OUT(n_i)_Q$ to denote the inflow and outflow of n_i associated with a callee Q . $IN(n_i)_Q$ denotes the flow of Q in all the paths arriving n_i , and $OUT(n_i)_Q$ denotes the flow of Q in all the paths arriving a successor of n_i through n_i .

Definition 4. The inflow of a basic block associated with Q is defined as:

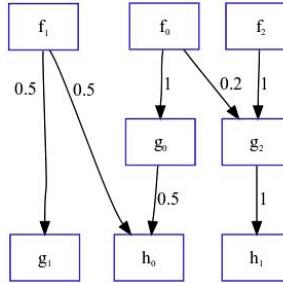
$$IN(n_i)_Q = \begin{cases} \sum_{e_{j,i} \in E} OUT(n_j)_Q & \text{if } (n_i \neq n_e \text{ and } n_i \text{ does not contain } Q) \\ 0 & \text{if } (n_i = n_e \text{ and } n_i \text{ does not contain } Q) \\ IN(n_i) & \text{if } (n_i \text{ contains } Q) \end{cases} \quad (3)$$

Definition 5. The outflow of a basic block associated with Q is defined as:

$$OUT(n_i)_Q = \begin{cases} IN(n_i)_Q & \text{if } (n_i \in N^S) \\ \frac{IN(n_i)_Q}{2} & \text{if } (n_i \in N^P) \end{cases} \quad (4)$$

In order to calculate the flow information effectively, we first rank the basic blocks using topological sorting. For n_e , we calculate $IN(n_e)_Q$ according to the definition. We then calculate the inflow of the basic blocks associated with Q in the order of topological sorting. If n_i contains Q , $IN(n_i)_Q$ is determined by $IN(n_i)$. Otherwise, $IN(n_i)_Q$ is the sum of the inflow of the predecessors of n_i associated with Q , which has been calculated. Finally, the flow of Q is equal to the inflow of n_x associated with Q , i.e., $IN(n_x)_Q$. The overall algorithm is shown in Algorithm 1.

Augmenting the Call Graph. We initialize the inflow of each procedure P by 1 (i.e., $n_0 = 1$), and propagate the flow in the CFG . Upon the completion of Algorithm 1, we

**Fig.3.** The FACG Example

augment every caller-callee edge according to $Q.Flow$ for each callee Q to build the FACG.

In the FACG, the significance of each caller-callee linkage can be indicated on each edge in the call graph. We revise the definition of the call graph [3] to give a formal definition of FACG.

Definition 6. A Flow-Augmented Call Graph (FACG) $G = (N, E)$ for procedure P is a directed multigraph in which each node $n \in N$ corresponds to either a caller P or a callee Q , and each weighted edge $e \in E$ represents a call-site augmented with control flow.

3.2 Recommending the Relevant APIs

The relevant APIs have similar functionalities, and thus they may access some program elements (i.e., the callee APIs and the structures) in common. Therefore, we adopt the set of program elements accessed by the APIs to recommend relevant APIs. However, an API may access many program elements (along with its callees), most of which are irrelevant to the main functionality of the API. These irrelevant elements can easily dominate the relevance calculation of APIs, and introduce noises to recommendation results⁶. To reduce the impact of the irrelevant elements, one may consider only the elements directly accessed by the APIs in the conventional call graph (For example, Saul *et al.* [19] consider only the neighboring functions of an API as the candidates for recommendation). But many relevant APIs for a API query may be far from the query in the call graph, and are missed by such kind of approaches.

Despite the difficulty of selecting a representative set of program elements accessed by an API from the conventional call graph, the task is feasible using the FACG. Given the FACG, the *significant* callees of a caller API can be found with regard to the flow-augmented edge. The representative set of program elements of an API is determined along with its *significant* callees. Basically, if a callee is called with a large flow in the FACG, it is considered to be the *significant* callee. Note that the flow can be propagated along the FACG, and a callee that is called indirectly by an API can be significant to the API as well. For example, in Fig. 3, f_0 calls g_0 with flow 1, and g_0 calls h_0 with

⁶ We further discuss this in section 4.4.

Table 2. Subject Project

Software	Version	KLOC	#C files	#Functions
Httpd	2.2.16	299.7	571	2188
D-Bus	1.1.3	99.2	108	1608
Tcl	8.5.9	227.1	207	1880
Tk	8.5.9	260.1	201	2303

flow 0.5, therefore, the flow of f_0 to h_0 is $1 * 0.5 = 0.5$. If we use 0.5 as the threshold of flow for determining *significant* callees, then the *significant* callees of f_0 , f_1 , and f_2 are: g_0, h_0 of f_0 ; g_1, h_0 of f_1 ; and g_2, h_1 for f_2 .

We then calculate the relation of two APIs as the cosine similarity of their representing vectors. Cosine similarity can capture the similarity of two vectors without biasing to vectors with large norm, and it is widely used in text retrieval. The cosine similarity of two vectors is

$$\text{cosine}(f, g) = \frac{f \cdot g}{\|f\| \cdot \|g\|} \quad (5)$$

For the example shown in Fig. 3, the relation of f_0 and f_1 is 0.5, and the relation of f_0 and f_2 is 0 (for simplicity of presentation, we omit the structures accessed by the APIs). While it is difficult to distinguish the relevance of f_1 and f_2 to f_0 in the conventional call graph, it is clear that f_1 is more relevant to f_0 than f_2 is in the FACG, since f_1 and f_0 have some main functionalities in common. We thus recommend f_1 as a highly relevant API of f_0 .

4 Evaluation

We compare the proposed approach with three state-of-the-art API recommendation tools: Suade [14], Fran [19] and Altair [8]. Suade recommends a set of API by analyzing the specificity and reinforcement, Fran performs a random walk algorithm in the call graph to find relevant APIs. Altair suggests the recommendation based on API's internal structural overlap. Our evaluation over the four tools is conducted with regard to the specific task suggested by Fran [19]: *Given a query API, retrieve other APIs in the same module*. The subject projects in our evolution section are Apache HTTP Server, Tcl/Tk library, and D-Bus message bus system. These subject projects are chosen because they are documented well, and the original API documentation which groups the APIs into modules is the best resource to tell the API relevance. Table 2 gives the basic description on the subject projects.

4.1 Experimental Setup

The experiments are conducted on an Intel Core 2 Duo 2.80GHz machine with 3GB memory and Linux 2.6.28 system. Suade, Altair, and Fran are freely available online. As mentioned in Altair [8], Suade is not initially designed for API recommendation; we use a re-implementation from Fran [19]. Fran proposes two algorithms, namely, FRAN

Table 3. A Comparison of API Recommendation Tools. Curly underline indicates a matched recommendation.

	Suade	Fran	Altair	our approach
apr_posix_file_get	N/A	N/A	apr_file_writev pipeblock pipenonblock proc_mutex_posix_cleanup proc_mutex_posix_acquire proc_mutex_posix_release proc_mutex_sysv_cleanup proc_mutex_sysv_acquire proc_mutex_sysv_release proc_mutex_fcntl_cleanup	apr_os_pipeput apr_os_pipeput_ex apr_file_pool_get apr_file_pool_put ~ apr_file_buffer_size_get apr_file_close apr_file_getetc apr_unix_child_file_cleanup apr_file_name_get apr_file_open_stderr
apr_fnmatch	tolower rangematch make_autoindex_entry	apr_palloc apr_pstrdup strlen apr_pstrcat ap_make_full_path ap_makedirstr_parent find_item toupper memset ignore_entry	N/A	rangematch pcre_maketables apruri_unparse apr_fnmatch_test ap_str_tolower atod strip_paren_comments ap_filter_protocol apr_match_glob ~ is_token
Tcl_SetVar2	Tcl_SetVar Tcl_SetVar2Ex EnvTraceProc TclipSetVariables Tcl_NewStringObj Tcl_SetString TclFreeObj	Tcl_SetVar TclPSetVariables EnvTraceProc TclExternalToUtfDString getuid uname _ctype_bloc Tcl_DStringInit TclGetPwUid Tcl_DStringFree	ObjFindNamespaceVar Tcl_FindNamespaceVar TclLookupSimpleVar TclObjJLookupVarEx TclObjJLookupVar TclLookupVar TclObjSetVar2 Tcl_SetVarEx Tcl_SetVar TclObjGetVar2	Tcl_SetVarEx Tcl_JnsSetVar2 Tcl_SetVarEx Tcl_GetVar2Ex ~ Tcl_SetVar2 TclUpVar2 TclVarErrMsg TclLookupVar Tcl_SetVar Tcl_SetVar TclObjJLookupVar Tcl_FindNamespaceVar

and FRIAR. They concludes that a combination of the two algorithms can achieve better performance. We use their implementation of the combined algorithm in our experiment, denoted as Fran to avoid the naming confusion. Suade and Fran need to be initialized with a call graph. We feed them with the call graph extracted by our implementation based on Gcc Gimple IR. Altair is built on LLVM, which can gracefully handle the source code. Moreover, Fran implements the top-k precision/recall measurement, we adopt the result to calculate the F1 score for Fran and Suade in our evaluation.

4.2 Case Studies

Case study is *de rigueur* in evaluating the result obtained by API recommenders [19]. Suade, Fran and Altair used human examination [14,19] and API naming [8] (concerned with the prefix `apr_` and `ap_` in Apache HTTP server). However, without convincing ground truth to support the judgement, these case studies have several limitations, which have been well-discussed in Fran [19] (which conducted an additional quantitative study as a supplement).

We sought to judge the result objectively and bring about fair comparisons among four tools. As suggested in Fran [19], the original project documentation which groups the APIs into modules is the best resource to judge the API relevance. Therefore, we take the module content as a yardstick to avoid subjective judgement on the correctness of relevant APIs and underline the relevant APIs that appear in the module in Table 3. Table 3 shows the recommendation set obtained by the four tools with respect to the queries list below. The cases are chosen in order to indicate the typical situations in the related projects.

Case 1: `apr_os_file_get()` is a function in the Apache Portable Runtime (APR) which the Apache HTTP server is built on top of. The APR documentation indicates that this API belongs to the Portability Routines module⁷ and its functionality is to “convert the file from apr type to os specific type”. This API is not directly called by Apache HTTP server, yet it exports the interface for developers to extend Apache HTTP server. In this case, Suade and Fran return no result with regard to this query because it is not in the call graph. Among the top 10 results returned by Altair, there is no relevant APIs according to the documentation. We investigate the source code and find that `apr_os_file_get()` is a simple function with two lines of code and accesses only one data structure `apr_status_t`. Altair computes the structural overlap among APIs; however, the useful information available for this query is limited and there are many APIs, which only access `apr_status_t`. Altair cannot distinguish the difference among them and returns the irrelevant results. Our approach investigates the structural information with the help of FACG and records how data structures are accessed by the APIs explicitly. Take `pipeblock()` in Altair’s result for example, this API accesses `apr_status_t` in different branches with regard to the control flow, where our approach is capable of distinguishing this case from a single access as in `apr_os_file_get()`. Among the top 10 results obtained by our approach, three APIs can be found in the documentation which are identified to be relevant to the query shown by a curly underline.

Case 2: `apr_fnmatch()` is a member of the Filename Matching Module in Apache HTTP server, which is described in the documentation as “to match the string to the given pattern”⁸. This query is a *self-contained* API, which simply manipulates the strings without accessing any data structures. As discussed in Altair [8], Altair may not return any result for these *self-contained* API. However, `apr_fnmatch()` is called by many other APIs in Apache HTTP server, and the documentation indicates that there are two APIs `apr_fnmatch_test()` and `apr_match_glob()`, that are relevant to it. Suade and Fran attempt to answer the query by searching the call graph. Since the call graph is not able to tell the significance of each callees to the caller. The result returned by Suade and Fran implies that those approaches rely on the conventional call graph may “get lost” in the API jungle because all the neighbour nodes in the call graph appear to be “the same”. Our approach, on the other hand, only considers the callee APIs on the major flow of the caller rather than those less important ones. Within our FACG, such explorations in the API jungle can be directed to the caller/callee more relevant to the query. In the end, our approach finds both of the other two APIs in the module according to the top 10 result.

Case 3: `Tcl_SetVar2()` in Tcl library belongs to the group of APIs that manipulate Tcl variables⁹. All of the four tools return meaningful results with regard to the query. `Tcl_SetVar2()` is widely used in Tcl to create/modify the variable, consequently, it has large neighbour sets (i.e., parent, child, sibling and spouse set defined by Fran). Both of Suade and Fran return `Tcl_SetVar()` which is a wrapper function of the query.

⁷ http://apr.apache.org/docs/apr/1.4/group_apr_portable.html

⁸ http://apr.apache.org/docs/apr/1.4/group_apr_fnmatch.html

⁹ <http://www.tcl.tk/man/tcl8.5/TclLib/SetVar.htm>

Table 4. The Precision and Recall Performance

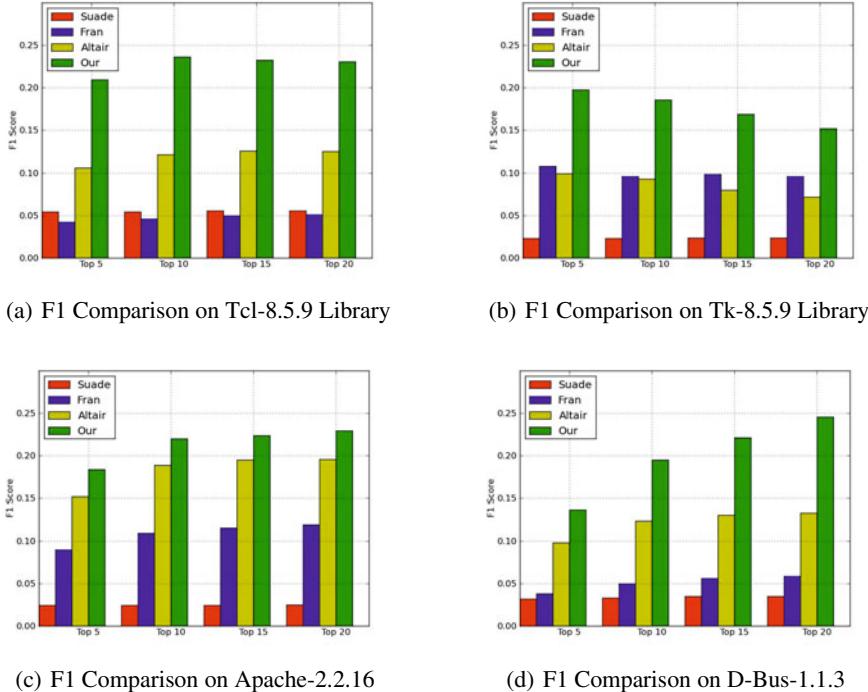
		Suade	Fran	Altair	Our approach
Precision	Top-5	0.111	0.155	0.266	0.384
	Top-10	0.104	0.134	0.236	0.319
	Top-15	0.109	0.135	0.222	0.278
	Top-20	0.109	0.135	0.212	0.252
Recall	Top-5	0.025	0.072	0.099	0.173
	Top-10	0.027	0.094	0.136	0.242
	Top-15	0.028	0.103	0.154	0.283
	Top-20	0.029	0.109	0.163	0.314
F1 Score	Top-5	0.033	0.070	0.114	0.181
	Top-10	0.034	0.075	0.132	0.209
	Top-15	0.035	0.080	0.133	0.213
	Top-20	0.035	0.081	0.132	0.213

Suade also returns `Tcl_SetVar2Ex()` which is called by the query. Most of the rest APIs in their results has nothing to do with variable manipulation. The situation is quite the same as in case 2; the conventional call graph does not distinguish the differences between callees. The top 10 results obtained by Altair and our approach are all related to variable manipulation in Tcl. This can be confirmed by taking look at the naming of these APIs. However, the relevant APIs (which are listed in the documentation and are the ground truth of F1 comparison) rank higher in our result. The reason behind is our approach supported by FACG is likely to consider the most important neighbours in the call graph. For example, query `Tcl_SetVar2` calls four APIs. Among them, `Tcl_SetVar2Ex` occupies the main flow of the query, so that it ranks high in the result. The result precisely illustrates the main advantage of the FACG. Moreover, in our top 10 results, we retrieve six out of the nine APIs that are documented in the same module, which is the best result from all four tools.

4.3 Quantitative Study

To perform the quantitative study, we compared the effectiveness of the four tools in retrieving relevant APIs at four recommendation-set size cutoffs, top-5, top-10, top-15 and top-20. Three measures (*precision*, *recall* and the F1-measure) of performance in information retrieval are adopted in our evaluation. All are defined by the recommendation set retrieved by the four tools. Let A be the recommended set obtained by each tools, and B be the set of relevant APIs which appear in the module. The *precision* and *recall* is defined as follows: $\text{precision} = |A \cap B|/|A|$ and $\text{recall} = |A \cap B|/|B|$. Precision measures the accuracy of obtaining the relevant APIs while recall measures the ability to obtain the relevant APIs. The F1-measure is the equally-weighted harmonic mean of the precision and recall measures, defined as $F = 2 * \text{precision} * \text{recall}/(\text{precision} + \text{recall})$. It is usually engaged as the combined measure of both precision and recall.

The summary of precision and recall performance is shown in Table 4. It can be seen that our approach achieves the highest precision. Moreover, our approach improves the precision rate over Suade, Fran and Altair by 184.8%, 120.6% and 31.7% respectively,

**Fig. 4.** Overall F1 Score Comparison

which indicates that our approach is able to suggest the most precise recommend set among the four. In addition, our approach achieves recall improvement over Suade, Fran and Altair by 828.4%, 167.7% and 83.3% respectively. Finally, the overall performance measurement is determined by F1-Score, where our approach achieves the highest F1 score with a large improvement of 495.6%, 166.7% and 59.7% over Suade, Fran and Altair respectively. The performance measurement indicates that our approach is able to recommend relevant APIs much more effectively than all other tools. Fig. 4 shows the F1 score comparisons over all the subject projects in our experiment. It is clearly seen that our approach dominates the performance in all recommendation-set size cutoffs.

4.4 Discussion on the Impact of FACG

The main insight in our work is to deploy the FACG to address the significance of caller-callee linkages. We conduct this subsection investigating our recommendation algorithm on the conventional call graph to further illustrate the impact of the FACG.

We apply our algorithm on the conventional call graph which treats every caller-callee linkage identically. Table 5 shows the F1 score of top-20 recommendation sets compared with our FACG approach on the four subject projects. On average, the performance of recommendation using the FACG is 41.7% higher than using the conventional call graph. More specifically, Table 6 lists the top 10 results of the case study in

Table 5. F1 Score of Top-20 Result

Subject Project	Httpd	D-bus	Tcl	Tk
Approach relying on the call graph	0.16	0.17	0.17	0.10
Approach relying on the FACG	0.23	0.24	0.23	0.15

Table 6. Top-10 results of our recommendation algorithm on the call graph

Query	apr_os_file_get	apr_fnmatch	Tcl_SetVar2
Top-10 Result	<u>apr_file_pool_get</u> <u>apr_file_buffer_size_get</u> <u>apr_file_unsetc</u> <u>apr_file_name_get</u> <u>apr_file_buffer_set</u> <u>apr_file_flush_locked</u> <u>apr_unix_child_file_cleanup</u> <u>database_cleanup</u> <u>apr_file_unlock</u> <u>apr_os_pipe_put_ex</u>	<u>rangematch</u> <u>find_desc</u> <u>make_parent_entry</u> <u>apr_match_glob</u> <u>apr_file_walk</u> <u>ap_location_walk</u> <u>ap_process_request_internal</u> <u>ap_process_resource_config</u> <u>include_config</u> <u>dummy_connection</u>	<u>Tcl_SetVar</u> <u>Tcl_SetVar2Ex</u> <u>TclTraceVar2</u> <u>TclResetResult</u> <u>TclObjSetVar2</u> <u>TclGetNamespaceForQualName</u> <u>TclTraceVar</u> <u>TclObjLookupVarEx</u> <u>EstablishErrorInfoTraces</u> <u>TclDeleteNamespace</u>

section 4.2. The relevant APIs supported by the module documentation are underlined with a curly line as well. As mentioned before, the recommendation algorithm based on the conventional call graph is blind to the difference between callees; therefore, it searches more candidates than our FACG approach, without distinguishing the significance of each candidate. Take `Tcl_SetVar2` for example, the first two results are directly linked with the query in call graph, thus they rank on the top in Table 6. However, although other relevant APIs (e.g., `Tcl_GetVar2`) appear in the candidate set, their significance is not clear enough in the call graph to be distinguished from other insignificant ones in the top 10 result. Moreover, the approach based on the call graph introduces some APIs (e.g., `TclGetNamespaceForQualName`) irrelevant to variable manipulation into the top 10 result, whereas the FACG approach can properly filter them as shown in Table 3. The evaluation demonstrates the benefit of the FACG in capturing the essence concerned with API usage and the impact of applying the FACG in API recommendation.

5 Related Work

There are mainly two categories of API recommendation approaches. The approaches which recommend APIs by using mining techniques belong to the first category. These approaches usually mine certain patterns or code snippets from sample code repositories. Prospector [9] developed by Mandelin *et al.* synthesizes the Jungloid graph to answer a query providing the input and output types. Prospector traverses the possible paths from input type to output type and recommends certain code snippets according to API signatures and a corpus of the client code. XSnippet [18] developed by Sahavechaphan *et al.* extends Prospector by adding more queries and ranking heuristics to mine code snippets from a sample repository. Context-sensitive is introduced to enhance the queries in XSnippet and produce more relevant results. Strathcona [4] developed by Holmes *et al.* is dedicated to recommending code examples matching the structural context. Six heuristics are applied to obtain the structural context description in the stored repository. MAPO [24] developed by Zhong *et al.* takes the advantage of mining frequent usage patterns of an API with the help of code search engines.

PARSEWeb [22] developed by Thummalapenta *et al.* mines open source repositories by using code search engines as well. Different from MAPO, PARSEWeb accepts the queries of the form “Source type⇒Destination type” and suggests the relevant methods that yield the object with the destination type.

The approaches in the second category aim at recommending APIs with respect to structural dependency. Zhang *et al.* propose a random-walk approach [23] based on PageRank algorithm to rank “popular” and “significant” program elements in Java programs. Inoue *et al.* proposed another approach [5] inspired by PageRank algorithm, which can be employed to rank valuable components in software systems based on the use relations. Suade [14] developed by Robillard is focused on providing suggestions for aiding program investigation. It accomplishes the suggestion by ranking the desired program elements concerned with the topological properties of structural dependency in software systems. Fran [19] developed by Saul *et al.* extends the topological properties by considering neighbouring relationships in the call graph. Altair [8] developed by Long *et al.* recommends the relevant APIs according to the overlap of commonly accessed variable information.

To the best of our knowledge, all of the previous API recommendation approaches rely on the conventional call graph. By distinguishing the significance of caller-callee linkages, the proposed FACG improves the accuracy of recommending relevant APIs.

6 Conclusion

This paper presents the Flow-Augmented Call Graph (FACG) to tame the API complexity. Augmenting the call graph by control flow analysis brings us a new foundation to capture the significance of caller-callee linkages. We employed API recommendation as a client application and engaged the FACG to retrieve the relevant APIs. We further conduct the experiment on four large projects with original documentation as ground truth to judge the performance, and compared our approach with three other state-of-the-art API recommendation tools. The case studies and quantitative evaluation results indicate our approach is more effective in retrieving the relevant APIs.

Acknowledgments. The authors would like to thank the anonymous reviewers for their insightful feedback. We would also like to thank Jie Zhang and Jianke Zhu with the writing. This research was fully supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK4154/10E).

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading (1986)
2. DeMarco, T., Lister, T.: Programmer performance and the effects of the workplace. In: ICSE, pp. 268–272 (1985)
3. Hall, M., Hall, M.W., Kennedy, K., Kennedy, K.: Efficient call graph analysis. ACM Letters on Programming Languages and Systems 1, 227–242 (1992)

4. Holmes, R., Murphy, G.C.: Using structural context to recommend source code examples. In: Inverardi, P., Jazayeri, M. (eds.) ICSE 2005. LNCS, vol. 4309, pp. 117–125. Springer, Heidelberg (2006)
5. Inoue, K., Yokomori, R., Fujiwara, H., Yamamoto, T., Matsushita, M., Kusumoto, S.: Component rank: Relative significance rank for software component search. In: ICSE, pp. 14–24 (2003)
6. Ko, A.J., Myers, B.A., Aung, H.H.: Six learning barriers in end-user programming systems. In: VL/HCC, pp. 199–206 (2004)
7. Lehman, M.M., Parr, F.N.: Program evolution and its impact on software engineering. In: ICSE, pp. 350–357 (1976)
8. Long, F., Wang, X., Cai, Y.: API hyperlinking via structural overlap. In: ESEC/SIGSOFT FSE, pp. 203–212 (2009)
9. Mandelin, D., Xu, L., Bodik, R., Kimelman, D.: Jungloid mining: helping to navigate the API jungle. In: PLDI, pp. 48–61 (2005)
10. Merrill, J.: Generic and Gimple: A new tree representation for entire functions. In: Proceedings of the 2003 GCC Developers Summit, Citeseer, pp. 171–179 (2003)
11. Podgurski, A., Clarke, L.A.: A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Transactions on Software Engineering 16, 965–979 (1990)
12. Ramanathan, M.K., Grama, A., Jagannathan, S.: Path-sensitive inference of function precedence protocols. In: ICSE, pp. 240–250 (2007)
13. Robillard, M., Walker, R., Zimmermann, T.: Recommendation systems for software engineering. IEEE Software 27(4), 80–86 (2010)
14. Robillard, M.P.: Automatic generation of suggestions for program investigation. In: ESEC/SIGSOFT FSE, pp. 11–20 (2005)
15. Robillard, M.P.: What makes APIs hard to learn? answers from developers. IEEE Software 26(6), 27–34 (2009)
16. Robillard, M.P., Coelho, W., Murphy, G.C.: How effective developers investigate source code: An exploratory study. IEEE Trans. Software Eng. 30(12), 889–903 (2004)
17. Ryder, B.G.: Constructing the call graph of a program. IEEE Trans. Software Eng. 5(3), 216–226 (1979)
18. Sahavechaphan, N., Claypool, K.T.: XSnippet: mining for sample code. In: OOPSLA, pp. 413–430 (2006)
19. Saul, Z.M., Filkov, V., Devanbu, P.T., Bird, C.: Recommending random walks. In: ESEC/SIGSOFT FSE, pp. 15–24 (2007)
20. Stylos, J., Myers, B.A.: Mica: A web-search tool for finding API components and examples. In: VL/HCC, pp. 195–202 (2006)
21. Tarjan, R.E.: Testing flow graph reducibility. In: STOC, pp. 96–107 (1973)
22. Thummalapenta, S., Xie, T.: PARSEWeb: a programmer assistant for reusing open source code on the web. In: ASE, pp. 204–213 (2007)
23. Zhang, C., Jacobsen, H.A.: Efficiently mining crosscutting concerns through random walks. In: AOSD, pp. 226–238 (2007)
24. Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H.: MAPO: mining and recommending api usage patterns. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 318–343. Springer, Heidelberg (2009)

Search-Based Design Defects Detection by Example

Marouane Kessentini¹, Houari Sahraoui¹, Mounir Boukadoum²,
and Manuel Wimmer³

¹ DIRO, Université de Montréal, Canada

{Kessentm, sahraouh}@iro.umontreal.ca

² DI, Université du Québec à Montréal, Canada

mounir.boukadoum@uqam.ca

³ Vienna University of Technology, Austria

wimmer@big.tuwien.ac.at

Abstract. We propose an automated approach to detect various types of design defects in source code. Our approach allows to automatically find detection rules, thus relieving the designer from doing so manually. Rules are defined as combinations of metrics/thresholds that better conform to known instances of design defects (defect examples). In our setting, we use and compare between different heuristic search algorithms for rule extraction: Harmony Search, Particle Swarm Optimization, and Simulated Annealing. We evaluate our approach by finding potential defects in two open-source systems. For all these systems, we found, in average, more than 75% of known defects, a better result when compared to a state-of-the-art approach, where the detection rules are manually or semi-automatically specified.

Keywords: Design defects, software quality, metrics, search-based software engineering, by example.

1 Introduction

Many studies report that software maintenance, traditionally defined as any modification made on a system after its delivery, consumes up to 90% of the total cost of a software project [2]. Adding new functionalities, correcting bugs, and modifying the code to improve its quality (by detecting and correcting design defects) are major parts of those costs [1]. There has been much research devoted to the study of bad design practices, also known in the literature as defects, antipatterns [1], smells [2], or anomalies [3]. Although bad practices are sometimes unavoidable, they should otherwise be prevented by the development teams and removed from the code base as early as possible in the design cycle.

Detecting and fixing defects is a difficult, time-consuming, and to some extent, manual process [5]. The number of outstanding software defects typically exceeds the resources available to address them [4]. In many cases, mature software projects are forced to ship with both known and unknown defects for lack of the development resources to deal with everyone. For example, in 2005, one Mozilla developer

claimed that “everyday, almost 300 bugs and defects appear . . . far too much for only the Mozilla programmers to handle”. To help cope with this magnitude of activity, several automated detection techniques have been proposed [5, 7].

The vast majority of existing work in defect detection relies on declarative rule specification [5, 7]. In these settings, rules are manually defined to identify the key symptoms that characterize a defect, using combinations of mainly quantitative, structural, and/or lexical indicators. However, in an exhaustive scenario, the number of possible defects to characterize manually with rules can be very large. For each defect, the metric combinations that serve to define its detection rule(s) require a substantial calibration effort to find the right threshold value to assign to each metric. Alternatively, [5] proposes to generate detection rules using formal definitions of defects. This partial automation of rule writing helps developers to concentrate on symptom description. Still, translating symptoms into rules is not obvious because there is no consensual symptom-based definition of design defects [8]. When a consensus exists, the same symptom could be associated with many defect types, which may compromise the precise identification of defect types. These difficulties explain a large portion of the high false-positive rates found in existing research [8].

The previous difficulties contrast with the ease of finding defect repositories in several companies, where defects that are manually identified and corrected are documented. This observation is at the origin of the work described herein. We start from the premise that defect repositories contain valuable information that can be used to mine regularities about defect manifestations, subsequently leading to the generation of detection rules. More concretely, we propose a new automated approach to derive rules for design defect detection. Instead of specifying rules manually for detecting each defect type, or semi-automatically using defect definitions, we extract them from valid instances of design defects. In our setting, we view the generation of design defect rules as an optimization problem, where the quality of a detection rule is determined by its ability to conform to a base of examples that contains instances of manually validated defects (classes).

The generation process starts from an initial set of rules that consists of random combinations of metrics. Then, the rules evolve progressively according to the set’s ability to detect the documented defects in the example base. Due to the potentially huge number of possible metric combinations that can serve to define rules, a heuristic approach is used instead of exhaustive search to explore the space of possible solutions. To that end, we use and compare between three rule induction heuristics : Harmony Search (HS) [9], Particle Swarm Optimization (PSO) [28] and Simulated Annealing (SA) [27] to find a near-optimal set of detection rules.

We evaluated our approach on defects present in two open source projects: GANTTPROJECT [11] and XERCES [12]. We used an n-fold cross validation procedure. For each fold, six projects are used to learn the rules, which tested on the remaining seventh project. Almost all the identified classes in a list of classes tagged as defects (blobs, spaghetti code and functional decomposition) in previous projects [17] were found, with a precision superior to 75%.

The remainder of this paper is structured as follows. Section 2 is dedicated to the problem statement. In Section 3, we describe the overview of our proposal. Then Section 4 describes the principles of the different heuristic algorithms used in our approach and the adaptations needed to our problem. Section 5 presents and discusses

the validation results. A summary of the related work in defect detection is given in Section 6. We conclude and suggest future research directions in Section 7.

2 Problem Statement

To understand better our contribution, it is important to define clearly the problem of defect detection. In this section, we start by giving the definitions of important concepts. Then, we detail the specific problems that are addressed by our approach.

2.1 Definitions

Design defects, also called design anomalies, refer to design situations that adversely affect the development of software. As stated by Fenton and Pfleeger [3], design defects are unlikely to cause failures directly, but may do it indirectly. In general, they make a system difficult to change, which may in turn introduce bugs.

Different types of defects, presenting a variety of symptoms, have been studied in the intent of facilitating their detection [1] and suggesting improvement paths. The two types of defects that are commonly mentioned in the literature are code smells and anti-patterns. In [2], Beck defines 22 sets of symptoms of common defects, named code smells. These include large classes, feature envy, long parameter lists, and lazy classes. Each defect type is accompanied by refactoring suggestions to remove it. Brown et al. [1] define another category of design defects, named anti-patterns, that are documented in the literature. In this section, we define the following three that will be used to illustrate our approach and in the detection tasks of our validation study.

- **Blob:** It is found in designs where one large class monopolizes the behavior of a system (or part of it), and other classes primarily encapsulate data.
- **Spaghetti Code:** It is a code with a complex and tangled control structure.
- **Functional Decomposition:** It occurs when a class is designed with the intent of performing a single function. This is found in code produced by non-experienced object-oriented developers.

For both types of defects, the initial authors focus on describing the symptoms to look for, in order to identify occurrences of these defects.

From the detection standpoint, the process consists of finding code fragments in the system that violate properties on internal attributes such as coupling and complexity. In this setting, internal attributes are captured through software metrics and properties are expressed in terms of valid values for these metrics [3]. The most widely used metrics are the ones defined by Chidamber and Kemerer [14]. These include the depth of inheritance tree DIT, weighted methods per class WMC and coupling between objects CBO. Variations of this metrics, adaptations of procedural ones as well as new metrics were also used such as the number of lines of code in a class LOCCLASS, number of lines of code in a method LOCMETHOD, number of attributes in a class NAD, number of methods NMD, lack of cohesion in methods LCOM5, number of accessors NACC, and number of private fields NPRIVFIELD.

2.2 Problem Statement

Although there is a consensus that it is necessary to detect design anomalies, our experience with industrial partners showed that there are many open issues that need to be addressed when defining a detection tool. Design anomalies have definitions at different levels of abstraction. Some of them are defined in terms of code structure, others in terms of developer/designer intentions, or in terms of code evolution. These definitions are in many cases ambiguous and incomplete. However, they have to be mapped into rigorous and deterministic rules to make the detection effective.

In the following, we discuss some of the open issues related to the detection.

How to decide if a defect candidate is an actual defect? Unlike software bugs, there is no general consensus on how to decide if a particular design violates a quality heuristic. There is a difference between detecting symptoms and asserting that the detected situation is an actual defect.

Are long lists of defect candidates really useful? Detecting dozens of defect occurrences in a system is not always helpful. In addition to the presence of false positives that may create a rejection reaction from development teams, the process of using the detected lists, understanding the defect candidates, selecting the true positives, and correcting them is long, expensive, and not always profitable.

What are the boundaries? There is a general agreement on extreme manifestations of design defects. For example, consider an OO program with a hundred classes from which one implements all the behavior and all the others are only classes with attributes and accessors. There is no doubt that we are in presence of a Blob. Unfortunately, in real life systems, we can find many large classes, each one using some data classes and some regular classes. Deciding which ones are Blob candidates depends heavily on the interpretation of each analyst.

How to define thresholds when dealing with quantitative information? For example, the Blob detection involves information such as class size. Although, we can measure the size of a class, an appropriate threshold value is not trivial to define. A class considered large in a given program/community of users could be considered average in another.

How to deal with the context? In some contexts, an apparent violation of a design principle is considered as a consensual practice. For example, a class Log responsible for maintaining a log of events in a program, used by a large number of classes, is a common and acceptable practice. However, from a strict defect definition, it can be considered as a class with abnormally large coupling.

In addition to these issues, the process of defining rules manually is complex, time-consuming and error-prone. Indeed, the list of all possible defect types can be very large. And each type requires specific rules.

To address or circumvent the above mentioned issues, we propose to use examples of manually found design defects to derive detection rules. Such examples are in general available and documents as part of the maintenance activity (version control logs, incident reports, inspection reports, etc.). The use of examples allows to derive rules that are specific to a particular company rather than rules that are supposed to be applicable to any context. This includes the definition of thresholds that correspond to the company best practices. Learning from examples aims also at reducing the list of detected defect candidates.

3 Approach Overview

This section shows how, under some circumstances, design defects detection can be seen as an optimization problem. We also show why the size of the corresponding search space makes heuristic search necessary to explore it.

3.1 Overview

We propose an approach that uses knowledge from previously manually inspected projects in order to detect design defects, called defects examples, to generate new detection rules based on a combinations of software quality metrics. More specifically, the detection rules are automatically derived by an optimization process that exploits the available examples.

Figure 1 shows the general structure of our approach. The approach takes as inputs a base of examples (*i.e.*, a set of defects examples) and a set of quality metrics, and generates as output a set of rules. The generation process can be viewed as the combination of metrics that best detect the defects examples. In other words, the best set of rules is that who detect the maximum number of defects.

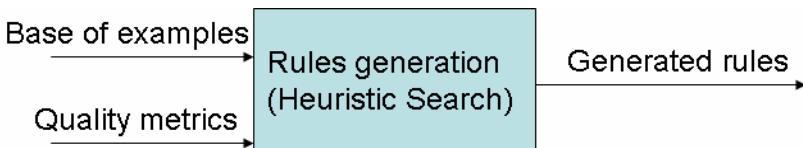


Fig. 1. Approach overview

As showed in Figure 2, the base of examples contains some projects (systems) that are inspected manually to detect all possible defects. In the training process, these systems are evaluated using the generated rules in each iterations of the algorithm. A fitness functions calculates the quality of the solution (rules) by comparing the list of detected defects with expected ones in the base.

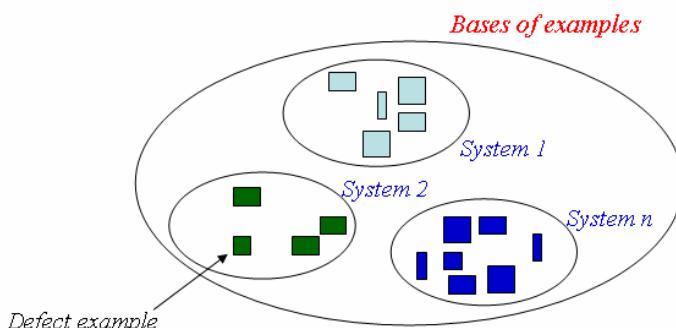


Fig. 2. Base of examples

As many metrics combinations are possible, the rules generation is a combinatorial optimization problem. The number of possible solutions quickly becomes huge as the number of metrics increases. A deterministic search is not practical in such cases, hence the use of heuristic search. The dimensions of the solution space are the metrics and some operators between them: union (metric1 OR metric2) and intersection (metric1 AND metric2). A solution is determined by the assignment of a threshold value to each metric. The search is guided by the quality of the solution according to the number of detected defects comparing to expected ones in the base.

To explore the solution space, we use different heuristic algorithms that will be detailed in Section 4.

3.2 Problem Complexity

Our approach assigns to each metric a corresponding threshold value. The number m of possible threshold value is very large. Furthermore, the rules generation process consists of finding the best combination between n metrics. In this context, $(n!)^m$ possible solutions have to be explored. This value can quickly become huge. A list of 5 metrics with 6 possible thresholds necessitates exploring at least 120^6 combinations. Considering these magnitudes, an exhaustive search cannot be used within a reasonable time frame. This motivates the use of a heuristic search if a more formal approach is not available or hard to deploy.

4 Search-Based Design Defect Detection by Example

We describe in this section the adaptation of three different heuristic algorithms to the design defects rules generation problem. To apply it to a specific problem, one must specify the encoding of solutions and the fitness function to evaluate a solution's quality. These two elements are detailed in subsections 4.1 and 4.2 respectively.

4.1 Solution Representation

One key issue when applying a search-based technique is finding a suitable mapping between the problem to solve and the techniques to use, *i.e.*, in our case, generating design defects rules. As stated in Section 3, we view the set of potential solutions as points in a n -dimensional space where each dimension corresponds to one metric or operator (union or intersection). Figure 3 shows an illustrative example which describes this rule: if (WMC ≥ 4) AND (TCC ≥ 7) AND (ATFD ≥ 1) Then Defect_Type(1)_detected. The WMC, TCC and ATFD are metrics defined as [14]:

- *Weighted Method Count* (WMC) is the sum of the statical complexity of all methods in a class. We considered the McCabe's cyclomatic complexity as a complexity measure.
- *Tight Class Cohesion* (TCC) is the relative number of directly connected methods.
- *Access to Foreign Data* (ATFD) represents the number of external classes from which a given class accesses attributes, directly or via accessor-methods.

- We used three types of defects : (1) blob, (2) spaghetti code and (3) functional decomposition.

The operator used as default is the intersection (and). The other operator (union) can be used as a dimension. The vector presented in Figure 3 generates only one rule. However, a vector may contain many rules separated by the dimension “Type”.

WMC	TCC	ATFD	Type
HigherThan(4)	HigherThan(7)	HigherThan(1)	Equal(1)

Fig. 3. Solution Representation

4.2 Evaluating Solutions

The *fitness function* quantifies the quality of the generate rules. As discussed in Section 3, the fitness function must consider the following aspect:

- Maximize the number of detected defects comparing to expected ones in the base of examples

In this context, we define the fitness function of a solution as

$$f = \sum_{i=1}^p a_i \quad (1)$$

Where p represents the number of detected classes. a_i has value 1 if the i th detected classes exists in the base of examples, and value 0 otherwise.

4.3 Search Algorithms

4.3.1 Harmony Search (HS)

The HS algorithm is based on natural musical performance processes that occur when a musician searches for a better state of harmony, such as during jazz improvisation [9]. Jazz improvisation seeks to find musically pleasing harmony as determined by an aesthetic standard, just as the optimization process seeks to find a global solution as determined by a fitness function. The pitch of each musical instrument determines the aesthetic quality, just as the fitness function value is determined by the set of values assigned to each dimension in the solution vector.

In general, the HS algorithm works as follows:

Step 1. Initialize the problem and algorithm parameters.

The HS algorithm parameters are specified in this step. They are the harmony memory size (HMS), or the number of solution vectors in the harmony memory; harmony memory considering rate (HMCR); bandwidth (bw); pitch adjusting rate (PAR); and the number of improvisations (K), or stopping criterion.

Step 2. Initialize the harmony memory.

The initial harmony memory is generated from a uniform distribution in the ranges $[x_{imin}, x_{imax}]$ ($i = 1, 2, \dots, N$) , as shown in Equation 1 :

$$HM = \begin{bmatrix} x_1^1 & x_2^1 & \dots & x_N^1 \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ x_1^{HMS} & x_2^{HMS} & \dots & x_N^{HMS-1} \end{bmatrix} \quad (2)$$

Step 3. Improvise a new harmony.

Generating a new harmony is called improvisation. The new harmony vector $x' = (x'_1, x'_2, \dots, x'_N)$ is determined by the memory consideration, pitch adjustment and random selection.

Step 4. Update harmony memory.

If the fitness of the improvised harmony vector $x' = (x'_1, x'_2, \dots, x'_N)$ is better than the worst harmony, replace the worst harmony in the IHM with x' .

Step 5. Check the stopping criterion: If the stopping criterion (maximum number of iterations K) is satisfied, computation is terminated. Otherwise, step 3 is repeated.

4.3.2 Particle Swarm Optimization (PSO)

PSO is a parallel population-based computation technique [31]. It was originally inspired from the flocking behavior of birds, which emerges from very simple individual conducts. Many variations of the algorithm have been proposed over the years, but they all share a common basis. First, an initial population (named swarm) of random solutions (named particles) is created. Then, each particle flies in the n -dimensional problem space with a velocity that is regularly adjusted according to the composite flying experience of the particle and some, or all, the other particles. All particles have fitness values that are evaluated by the objective function to be optimized. Every particle in the swarm is described by its position and velocity. A particle position represents a possible solution to the optimization problem, and velocity represents the search distances and directions that guide particle flying. In this paper, we use basic velocity and position update rules defined by [31]:

$$V_{id} = W * V_{id} + C_1 * rand() * (P_{id} - X_{id}) + C_2 * Rand() * (P_{gd} - X_{id}) \quad (3)$$

$$X_{id} = X_{id} + V_{id} \quad (4)$$

At each time (iteration), V_{id} represents the particle velocity and X_{id} its position in the search space. P_{id} (also called *pbest* for local best solution), represents the i^{th} particle's best previous position, and P_{gd} (also called *gbest* for global best solution), represents the best position among all particles in the population. w is an inertia term; it sets a balance between the global and local exploration abilities in the swarm. Constants c_1 and c_2 represent cognitive and social weights associated to the individual and global behavior, respectively. There are also two random functions *rand()* and *Rand()*

(normally uniform in the interval [0, 1]) that represent stochastic acceleration during the attempt to pull each particle toward the $pbest$ and $gbest$ positions. For a n -dimensional search space, the i^{th} particle in the swarm is represented by a n -dimensional vector, $\mathbf{x}_i=(x_{i1},x_{i2},\dots,x_{id})$. The velocity of the particle, $pbest$ and $gbest$ are also represented by n -dimensional vectors.

4.3.3 Simulated Annealing (SA)

SA [19] is a search algorithm that gradually transforms a solution following the annealing principle used in metallurgy.

After defining an initial solution, the algorithm iterates the following three steps:

- 1 Determine a new neighboring solution,
- 2 Evaluate the fitness of the new solution
- 3 Decide on whether to accept the new solution in place of the current one based on the fitness gain/lost ($\Delta cost$).

When $\Delta cost < 0$, the new solution has lower cost than the current solution and it is accepted. For $\Delta cost > 0$ the new solution has higher cost. In this case, the new solution is accepted with probability $e^{-\Delta cost/T}$. The introduction of a stochastic element in the decision process avoids being trapped in a local minimum solution. Parameter T , called temperature, controls the acceptance probability of a lesser good solution. T begins with a high value, for a high probability of accepting a solution during the early iterations. Then, it decreases gradually (cooling phase) to lower the acceptance probability as we advance in the iteration sequence. For each temperature value, the three steps of the algorithm are repeated for a fixed number of iterations.

5 Validation

To test our approach, we studied its usefulness to guide quality assurance efforts on an open-source program. In this section, we describe our experimental setup and present the results of an exploratory study.

5.1 Goals and Objectives

The goal of the study is to evaluate the efficiency of our approach for the detection of design defects from the perspective of a software maintainer conducting a quality audit. We present the results of the experiment aimed at answering the following research questions:

RQ1: To what extent can the proposed approach detect design defects?

RQ2: What types of defects does it locate?

To answer RQ1, we used an existing corpus of known design defects to evaluate the precision and recall of our approach. We compared our results to those produced by an existing rule-based strategy [5]. To answer RQ2, we investigated the type of defects that were found.

5.2 System Studied

We used two open-source Java projects to perform our experiments: GanttProject (Gantt for short) v1.10.2, and Xerces-J v2.7.0.

Table 1. Program statistics

Systems	Number of classes	KLOC
GanttProject v1.10.2	245	31
Xerces-J v2.7.0	991	240

Table 1 summarizes facts on these programs. Gantt is a tool for creating project schedules by means of Gantt charts and resource-load charts. Gantt enables breaking down projects into tasks and establishing dependencies between these tasks. Xerces-J is a family of software packages for parsing and manipulating XML. It implements a number of standard APIs for XML parsing.

In our experiments, we used some of the classes in Gantt as our example set of design defects. These examples are validated manually by a group of experts [17]. We choose the Xerces-J and Gantt libraries because they are medium sized open-source projects and were analysed in related work. The version of Gantt studied was known to be of poor quality, which lead to a new major version. Xerces-J on the other hand has been actively developed over the past 10 years and its design has not been responsible for a slowdown of its development.

In [5], Moha et al. asked three groups of students to analyse the libraries to tag instances of specific antipatterns to validate their detection technique, DECOR. For replication purposes, they provided a corpus of describing instances of different antipatters including: Blob classes, Spaghetti code, and Functional Decompositions. Blobs are classes that do or know too much. Spaghetti Code (SC) is code that does not use appropriate structuring mechanisms. Functional Decomposition (FD) is code that is structured as a series of function calls. These represent different types of design risks. In our study, we verified the capacity of our approach to locate classes that corresponded to instances of these antipatterns. Thus, Xerces-J is then analyzed using some defects examples from Gantt and vice-versa.

The obtained results were compared to those of DECOR [5]. For every antipattern in Xerces-J and Gantt, they published the number of antipatterns detected, the number of true positives, the recall (number of true positives over the number of design defects) and the precision (ratio of true positives over the number detected). Our comparison is consequently done using precision and recall.

5.3 Results

Tables 2, 3 and 4 summarize our findings. The results show that HS performs comparing to PSO and SA. In fact, the two global search algorithms HS and PSO are suitable to explore large search space. For Gantt, our precision average is 87%. DECOR on the other hand has a combined precision of 59% for its detection on the

Table 2. HS results

System	Precision	Recall
Gantt	Spaghetti: 82% Blob: 100% F.D: 87%	Spaghetti: 90% Blob: 100% F.D: 47%
Xerces-J	Spaghetti:82% Blob:93% F.D:76%	Spaghetti:84% Blob:94% F.D:60%

Table 3. PSO results

System	Precision	Recall
Gantt	Spaghetti: 79% Blob: 100% F.D: 82%	Spaghetti: 94% Blob: 100% F.D: 53%
Xerces-J	Spaghetti:89% Blob:91% F.D:73%	Spaghetti:81% Blob:92% F.D:68%

Table 4. SA results

System	Precision	Recall
Gantt	Spaghetti: 81% Blob: 100% F.D: 80%	Spaghetti: 95% Blob: 100% F.D: 51%
Xerces-J	Spaghetti:77% Blob:91% F.D:71%	Spaghetti:80% Blob:92% F.D:69%

same set of antipatterns. For Xerces-J, our precision average is of 83%. For the same dataset, DECOR had a precision of 67%. However, the recall score for both systems is less than DECOR. In fact, the rules defined in DECOR are large and this is explained by the lower score in terms of precision, In the context of this experiment, we can conclude that our technique is able to accurately identify design anomalies more accurately than DECOR (RQ1).

We noticed that our technique does not have a bias towards the detection of specific anomaly types. In Xerces-J, we had an almost equal distribution of each antipattern. On Gantt, the distribution is not as balanced. This is principally due to the number of actual antipatterns in the system.

The detection of FDs using only metrics seems difficult. This difficulty is why DECOR includes an analysis of naming conventions to perform its detection. Using naming convention means that their results depend on the coding practices of a development team. Our results are however comparable to theirs while we do not leverage lexical information. The complete results of our experiments, including the comparison with DECOR, can be found in [18].

5.4 Discussion

The reliability of the proposed approach requires an example set of bad code. It can be argued that constituting such a set might require more work than identifying, specifying, and adapting rules. In our study, we showed that by using Gantt or Xerces-J directly, without any adaptation, the technique can be used out of the box and this will produce good detection and recall results for the detection of antipatterns for the two systems studied.

The performance of this detection was superior to that of DECOR. In an industrial setting, we could expect a company to start with Xerces-J or Gantt, and gradually migrate its set of bad code examples to include context-specific data. This might be essential if we consider that different languages and software infrastructures have different best/worst practices.

Another issue is the rules generation process. The detection results might vary depending on the used rules which are generated randomly though guided by a meta-heuristic. To ensure that our results are relatively stable, we compared the results of multiple executions for rules generation. We observed an average recall and precision more than 80% for both Gantt and Xerces-J with the three different heuristic search algorithms. Furthermore, we found that the majority of defects detected are found in every execution. We consequently believe that our technique is stable, since the precision and recall scores are approximately the same for different executions.

Another important advantage comparing to machine learning techniques is that our search algorithms do not need both positive (good code) and negative (bad code) examples to generate rules like for example Inductive Logic Programming [19].

Finally, since we viewed the design defects detection problem as a combinatorial problem addressed with heuristic search, it is important to contrast the results with the execution time. We executed our algorithm on a standard desktop computer (Pentium CPU running at 2 GHz with 2GB of RAM). The execution time for rules generation with a number of iteration (stopping criteria) fixed to 500 is less than three minutes (2min36s). This indicates that our approach is scalable from the performance standpoint. However, the execution time depends to the number of used metrics and the size of the base of examples. It should be noted that more important execution times may be obtained in comparison with using DECOR. In any case, our approach is meant to apply to situations where manual rule-based solutions are normally not easily available.

6 Related Work

Several studies have recently focused on detecting design defects in software using different techniques. These techniques range from fully automatic detection to guided manual inspection. The related work can be classified into three broad categories: metric-based detection, detection of refactoring opportunities, visual-based detection.

In first category, Marinescu [7] defined a list of rules relying on metrics to detect what he calls design flaws of OO design at method, class and subsystem levels. Erni et al. [20] use metrics to evaluate frameworks with the goal of improving them. They introduce the concept of multi-metrics, as an n-tuple of metrics expressing a quality

criterion (e.g., modularity). The main limitation of the two previous contributions is the difficulty to define manually threshold values for metrics in the rules. To circumvent this problem, Alikacem et al. [21] express defect detection as fuzzy rules with fuzzy label for metrics, e.g., small, medium, large. When evaluating the rules, actual metric values are mapped to truth value for the labels by means of membership functions. Although no thresholds have to be defined, still, it is not obvious to decide for membership functions.

The previous approaches start from the hypothesis that all defect symptoms could be expressed in terms of metrics. Actually, many defects involve notions that could not be quantified. This observation was the foundation of the work of Moha et al. [5]. In their approach, named DECOR, they start by describing defect symptoms using an abstract rule language. These descriptions involve different notions such as class roles and structures. The descriptions are later mapped to detection algorithms. In addition to the threshold problem, this approach uses heuristics to approximate some notions with results in an important rate of false positives. Khomh et al. [4] extended DECOR to support uncertainty and to sort the defect candidates accordingly. Uncertainty is managed by Bayesian belief networks that implement the detection rules of DECOR. The detection outputs are probabilities that a class is an occurrence of a defect type.

In our approach, all the above mentioned problems related to the use of rules and metrics do not arise. Indeed, the symptoms are not explicitly used, which reduces the manual adaptation/calibration effort.

In the second category of work, defects are not detected explicitly. They are implicitly because, the approaches refactor a system by detecting elements to change to improve the global quality. For example, in [22], defect detection is considered as an optimization problem. The authors use a combination of 12 metrics to measure the improvements achieved when sequences of simple refactorings are applied, such as moving methods between classes. The goal of the optimization is to determine the sequence that maximizes a function, which captures the variations of a set of metrics [23]. The fact that the quality in terms of metrics is improved does not necessarily mean that the changes make sense. The link between defect and correction is not obvious, which makes the inspection difficult for the maintainers. In our case, we separate the detection and correction phase. In [8, 26], we have proposed an approach for the automatic detection of potential design defects in code. The detection is based on the notion that the more code deviates from good practices, the more likely it is bad. Taking inspiration from artificial immune systems, we generated a set of detectors that characterize different ways that a code can diverge from good practices. We then used these detectors to measure how far code in assessed systems deviates from normality.

7 Conclusion

In this article, we presented a novel approach for tackling the problem of detecting design defects. Typically, researchers and practitioners try to characterize different types of common design defects and present symptoms to use in order to locate them in a system. In our work, we show that we do not need this knowledge to perform detection. Instead, all we need is some examples of design defects to generate

detection rules. Interestingly enough, our study shows that our technique outperforms DECOR [5], a state of the art, metric-based approach, where rules are defined manually, on its test corpus.

The proposed approach was tested on open-source systems and the results were promising. The detection process uncovered different types of design defects was more efficiently than DECOR. The comparison between three heuristic algorithm shows that HS give better results than PSO and SA. Furthermore, as DECOR needed an expert to define rules, our results were achieved without any expert knowledge, relying only on the bad structure of Gantt to guide the detection process.

The benefits of our approach can be summarized as follows: 1) it is fully automatable; 2) it does not require an expert to manually write rules for every defect type and adapt them to different systems; 3) the rule generation process is executed once; then, the obtained rules can be used to evaluate any system.

The major limitations of our approach are: 1) the generated rules are based on metrics, and some defects may require additional or different knowledge to be detected; 2) the approach requires the availability of a code base that is representative of bad design practices, and where all the possible design defects are already detected.

As part of our future work, we plan to explore the second step: correction of detected design defects (refactoring). Furthermore, we need to extend our base of examples with other bad-designed code in order to take into consideration different programming contexts.

References

1. Brown, W.J., Malveau, R.C., Brown, W.H., McCormick III, H.W., Mowbray, T.J.: *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st edn. John Wiley and Sons, Chichester (March 1998)
2. Fowler, M.: *Refactoring – Improving the Design of Existing Code*. 1st edn. Addison-Wesley, Reading (June 1999)
3. Fenton, N., Pfleeger, S.L.: *Software Metrics: A Rigorous and Practical Approach*, 2nd edn. International Thomson Computer Press, London (1997)
4. Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., Sahraoui, H.: A Bayesian Approach for the Detection of Code and Design Smells. In: Proc. of the ICQS 2009 (2009)
5. Moha, N., Guéhéneuc, Y.-G., Duchien, L., Meur, A.-F.L.: DECOR: A method for the specification and detection of code and design smells. *Transactions on Software Engineering (TSE)*, 16 pages (2009)
6. Liu, H., Yang, L., Niu, Z., Ma, Z., Shao, W.: Facilitating software refactoring with appropriate resolution order of bad smells. In: Proc. of the ESEC/FSE 2009, pp. 265–268 (2009)
7. Marinescu, R.: Detection strategies: Metrics-based rules for detecting design flaws. In: Proc. of ICM 2004, pp. 350–359 (2004)
8. Kessentini, M., Vaucher, S., Sahraoui, H.: Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In: Proc. of the International Conference on Automated Software Engineering, ASE 2010 (2010)
9. Lee, K.S., Geem, Z.W.: A new meta-heuristic algorithm for continuous engineering optimization: harmony search theory and practice. *Comput. Method Appl. M* 194(36-38), 3902–3933 (2005)

10. Lee, K.S., Geem, Z.W., Lee, S.H., Bae, K.W.: The harmony search heuristic algorithm for discrete structural optimization. *Eng Optimiz* 37(7), 663–684 (2005)
11. <http://ganttpoint.project.biz/index.php>
12. <http://xerces.apache.org/>
13. Riel, A.J.: Object-Oriented Design Heuristics. Addison-Wesley, Reading (1996)
14. Gaffney, J.E.: Metrics in software quality assurance. In: Proc. of the ACM 1981 Conference, pp. 126–130. ACM, New York (1981)
15. Mantyla, M., Vanhanen, J., Lassenius, C.: A taxonomy and an initial empirical study of bad smells in code. In: Proc. of ICSM 2003. IEEE Computer Society, Los Alamitos (2003)
16. Wake, W.C.: Refactoring Workbook. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
17. http://www.ptidej.net/research/decor/index_html
18. <http://www.marouane-kessentini/FASE10.zip>
19. Raedt, D.: Advances in Inductive Logic Programming, 1st edn. IOS Press, Amsterdam (1996)
20. Erni, K., Lewerentz, C.: Applying design metrics to object-oriented frameworks. In: Proc. IEEE Symp. Software Metrics. IEEE Computer Society Press, Los Alamitos (1996)
21. Alikacem, H., Sahraoui, H.: Détection d'anomalies utilisant un langage de description de règle de qualité, in actes du 12e colloque LMO (2006)
22. O'Keeffe, M., Cinnéide, M.: Search-based refactoring: an empirical study. *Journal of Software Maintenance* 20(5), 345–364 (2008)
23. Harman, M., Clark, J.A.: Metrics are fitness functions too. In: IEEE METRICS, pp. 58–69. IEEE Computer Society Press, Los Alamitos (2004)
24. Kessentini, M., Sahraoui, H.A., Boukadoum, M.: Model Transformation as an Optimization Problem. In: Busch, C., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 159–173. Springer, Heidelberg (2008)
25. Kirkpatrick, D.S., Gelatt, Jr., Vecchi, M.P.: Optimization by simulated annealing. *Science* 220(4598), 671680 (1983)
26. Eberhart, R.C., Shi, Y.: Particle swarm optimization: developments, applications and resources. In: Proc. IEEE Congress on Evolutionary Computation (CEC 2001), pp. 81–86 (2001)

An Empirical Study on Evolution of API Documentation

Lin Shi¹, Hao Zhong¹, Tao Xie³, and Mingshu Li^{1,2}

¹ Laboratory for Internet Software Technologies, Institute of Software,
Chinese Academy of Sciences, Beijing 100190, China

² Key Laboratory for Computer Science, Chinese Academy of Sciences, Beijing 100190, China

³ Department of Computer Science, North Carolina State University, USA
`{shilin,zhonghao}@itechs.iscas.ac.cn, xie@csc.ncsu.edu,`
`mingshu@iscas.ac.cn`

Abstract. With the evolution of an API library, its documentation also evolves. The evolution of API documentation is common knowledge for programmers and library developers, but not in a quantitative form. Without such quantitative knowledge, programmers may neglect important revisions of API documentation, and library developers may not effectively improve API documentation based on its revision histories. There is a strong need to conduct a quantitative study on API documentation evolution. However, as API documentation is large in size and revisions can be complicated, it is quite challenging to conduct such a study. In this paper, we present an analysis methodology to analyze the evolution of API documentation. Based on the methodology, we conduct a quantitative study on API documentation evolution of five widely used real-world libraries. The results reveal various valuable findings, and these findings allow programmers and library developers to better understand API documentation evolution.

1 Introduction

In modern software industries, it is a common practice to use Application Programming Interface (API) libraries (*e.g.*, J2SE¹) to assist development, and API documentation is typically shipped with these API libraries. With API documentation, library developers provide documents on functionalities and usages of API elements (*i.e.*, classes, methods, and fields of API libraries), and programmers of library API client code (referred to as programmers for short in this paper) follow these documents to use API elements.

Due to various factors such as adding new functionalities and improving API usability, both API libraries and their documentation evolve across versions. For example, the document of the `java.sql.connection.close()` method in J2SE 1.5² has a notice that connections can be automatically closed without calling the `close` method (Figure 1a). In J2SE 1.6³, library developers delete the notice, and emphasize the importance of calling the `close` method explicitly (Figure 1b). In practice, the preceding document of J2SE 1.5 is misleading, and causes many related defects. For example, a known defect⁴ of the Chukwa project is related to unclosed JDBC connections. Existing

¹ <http://www.oracle.com/technetwork/java/javase/overview>

² <http://java.sun.com/j2se/1.5.0/docs/api/>

³ <http://java.sun.com/javase/6/docs/api/>

⁴ <http://issues.apache.org/jira/browse/CHUKWA-9>

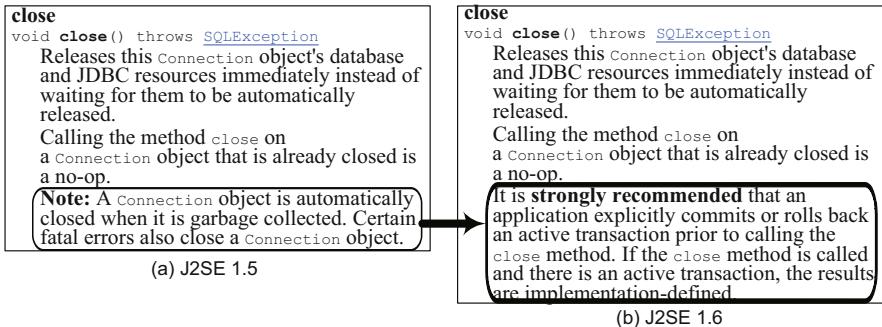


Fig. 1. An example of API documentation evolution

research [16] shows that programmers are often unwilling to read API documentation carefully. If programmers miss the revision in J2SE 1.6, they may follow the old document in J2SE 1.5, and still introduce defects that are related to unclosed connections even after the document is modified. From the revision, library developers can also learn a lesson, since their API documentation contains misleading documents. In summary, revisions in API documentation are important for both client code development and library development.

As API documentation contains documents for hundreds or even thousands of API elements, revisions in API documentation could also be quite large in size. Due to the pressure of software development, it is difficult for programmers and library developers to analyze these revisions systematically and comprehensively. A quantitative study on evolution of API documentation can help programmers and library developers better understand the evolution, so there is a strong need of such a quantitative study. However, to the best of our knowledge, no previous work presents such a study, since textual revisions across versions are typically quite large, and analyzing these revisions requires a large amount of human effort and a carefully designed analysis methodology.

In this paper, we present an analysis methodology to analyze API documentation evolution quantitatively. Based on the methodology, we conduct a quantitative study on documentation provided by five widely used real-world libraries.

The main contributions of this paper are as follows:

- We highlight the importance of API documentation evolution, and propose a methodology to analyze the evolution quantitatively.
- Based on our methodology, we provide the first quantitative analysis on API documentation evolution. The results show various aspects of API documentation evolution. The results allow programmers and library developers to better understand API documentation evolution quantitatively.

The rest of this paper is organized as follows. Section 2 presents our analysis methodology. Section 3 presents our empirical results. Section 4 discusses issues and future work. Section 5 introduces related work. Section 6 concludes.

2 Analysis Methodology

To conduct a systematic and quantitative study on API documentation, we present a methodology to analyze documentation evolution. Given API documentation of two versions of an API library, we classify their revisions into various categories.

Step 1: Identifying revisions. In this step, we first compare API documents of the two versions to find their revisions. In API documentation, we refer to a collection of words that describe an API element as a document. During evolution, an API element may be added, removed, and modified (see Section 3.3 for our results). As we focus on API documentation, we consider an API element modified when its declaration changes or its document changes. For a changed document, we refer to a pair of two associated sentences with differences as a revision. We divide a revision into one of the three categories: (1) an addition: a newly added sentence; (2) a deletion: a deleted sentence; (3) a modification: a modified sentence. Finally, we extract various characteristics such as word appearances and change locations from identified revisions.

Step 2: Classifying revisions based on heuristic rules. In this step, we first analyze a few hundred randomly selected sample revisions, and extract characteristics for various categories manually. For each category, we define heuristic rules according to these characteristics. For example, we find that different types of annotations (*e.g.*, @see and @version) are associated with different key words with specific fonts, so we define a corresponding heuristic rule to classify revisions by their annotations. With these heuristic rules, we classify revisions into various clusters by our heuristic rules. Revisions in the same cluster share similar characteristics.

Step 3: Refining and analyzing classified revisions. In this step, we tune our classifiers iteratively. In each iteration, we analyze inappropriately classified results to refine existing heuristic rules or to implement new rules for better classification. When classified results are accurate enough, we further fix remaining inappropriately classified results, and analyze results for insights on the evolution of API documentation (see Sections 3.1 and 3.2 for our results).

3 Empirical Study

In our empirical study, we focus on three research questions as follows:

RQ1: Which parts of API documentation are frequently revised?

RQ2: To what degree do such revisions indicate behavioral differences?

RQ3: How frequently are API elements and their documentation changed?

We use five widely used real-world libraries as subjects. The five libraries have 9,506, 580 words of API documentation in total. For each library, we analyze the latest five stable versions as shown in Table 1. For J2SE, we do not choose some end-of-life versions (*i.e.*, J2SE 1.4.1, J2SE 1.4.0, and J2SE 1.3.0)⁵. We still choose J2SE 1.2.2 (an end-of-life version), since existing stable versions of J2SE do not have five releases.

⁵ <http://java.sun.com/products/archive/j2se-eol.html>

Table 1. Versions of selected libraries

Library	v1	v2	v3	v4	v5
J2SE	1.2.2	1.3.1	1.4.2	1.5	1.6
ActiveMQ	5.0.0	5.1.0	5.2.0	5.3.0	5.3.1
lucene	2.9.0	2.9.1	2.9.2	3.0.0	3.0.1
log4j	1.2.12	1.2.13	1.2.14	1.2.15	1.2.16
struct	2.0.14	2.1.2	2.1.6	2.1.8	2.1.8.1

For ActiveMQ, we analyze its core API elements only. More details can be found on our project site: <https://sites.google.com/site/asergroup/projects/apidocevolution>.

3.1 RQ1: Which Parts of API Documentation Are Frequently Revised?

In this section, we present proportions of all types and some examples of these types. In total, we compared 2,131 revisions that cover the `java.util` package of J2SE, and all the other four libraries. We identify three primary categories of API documentation evolution, and for each category, we further identify detailed revision types, as shown in Figure 2. The vertical axis shows the three primary categories of revisions, and the horizontal axis shows the proportion for each category over the 2,131 revisions.

Finding 1: 45.99% of revisions are about annotations as follows.

In the official guidance of Java documentation⁶, tags and annotations are different. By using tags, library developers can add structures and contents to the documentation (*e.g.*, `@since`), whereas annotations are used to affect the way API elements are treated by tools and libraries (*e.g.*, `@Entity`). In this paper, we do not distinguish the two definitions and use *annotation* to represent both of them for simplicity.

Version 19.25% A version consists of numbers and dates that indicate when an API element is created or changed, and a version is marked by the `@version` annotation or the `@since` annotation. With either annotation, the version of an API element is updated automatically when API code changes. As the two annotations cause to automatically modify documents, they cause a large proportion of revisions. It is still an open question on whether such version numbers are useful, since we notice that library developers systematically deleted the two annotations in the documentation of lucene 3.0.0.

Exception 8.21% Exception handling plays an important role in the Java language, and the exceptions of an API method can be marked by the `@throws` annotation or the `@exception` annotation. Revisions on exceptions often indicate behavioral differences, and Finding 4 in Section 3.2 presents more such examples.

Reference 7.60% The document of one API element may refer programmers to other API elements since these API elements are related. The reference relations can be marked by the `@see` annotation or the `@link` annotation. We find that at least two factors drive library developers to modify reference relations. One factor is that the

⁶ <http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>

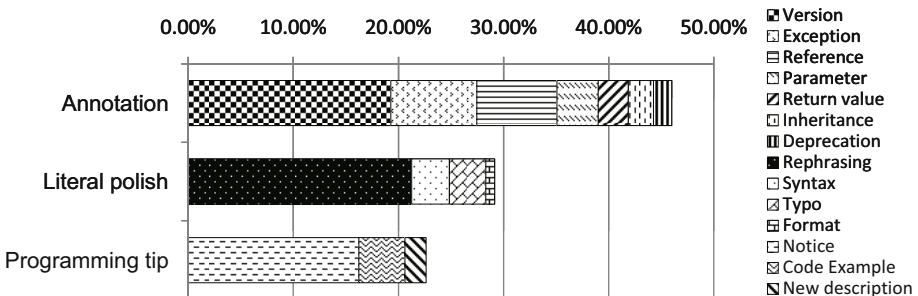


Fig. 2. Categories of revisions

names of referred API elements are modified. The other factor is that it is difficult even for library developers to decide reference relations among API elements. Besides other API elements, one document may even refer to some Internet documents using URLs. Library developers may also modify URLs across versions. For example, in the `java.util.Locale.getISO3Language()` method of J2SE 1.5, the URL of ISO 639-2 language codes is updated from one⁷ to another⁸.

Using the annotations such as `@see` or `@link`, one document may refer to other code elements or documents, and these code elements or documents may get updated across versions. Code refactoring [15] is a hot research topic, but most of existing refactoring approaches address the problem of refactoring code partially, and its impacts on documents are less exploited.

Parameter 3.94% For each API method, the document of its parameters can be marked by the `@param` annotation, and library developers may add parameter documents to describe parameters. For example, in J2SE 1.3.1, library developers add a document to the `comp` parameter of the `java.util.Collections.max(Collection, Comparator)` method: “`comp` the comparator with which to determine the maximum element. A null value indicates that...”.

Return value 2.86% The return value of an API method can be marked by the `@return` annotation, and library developers may add return-value documents to better describe return values. For example, in J2SE 1.4.2, library developers add a document to the `java.util.ArrayList.contains(Object)` method: “Return: `true` if the specified element is present; `false` otherwise”.

Inheritance 2.39% Inheritances among classes and interfaces cause similar documents across them, and also their methods and fields. The document of a class often needs to be modified, when the document of its superclass is modified. Although library developers can use the `@inheritDoc` annotation to deal with similar documents caused by inheritance relations, we notice that only a small proportion of such similar documents are marked by the annotation.

Deprecation 1.74% An API element may become deprecated, and its document can be marked by the `@deprecated` annotation. When the document of an API element

⁷ <ftp://dkuug.dk/i18n/iso-639-2.txt>

⁸ <http://www.loc.gov/standards/iso639-2/englangn.html>

is marked as deprecated, some IDEs such as the Eclipse IDE explicitly warn that programmers should be careful to use the API element. When an API element becomes deprecated, library developers sometimes may suggest programmers to use alternative API elements. Although a deprecated API element is assumed to be deleted in later versions, we find that some deprecated API elements can become undeprecated again. For example, we find that eight API elements (*e.g.*, the `org.apache.lucene.search.function.CustomScoreQuery.customExplain()` method) get updated from deprecated to undeprecated in lucene 3.0.0.

Implication 1: Library developers take much effort to write annotations, and some annotations (*e.g.*, `@see`) are difficult to maintain. Some tools may be beneficial if they can help library developers write and update these annotations. Researchers may borrow ideas from existing research on code refactoring when implementing these tools.

Finding 2: 29.14% of revisions are literal polishes as follows.

Rephrasing 21.26% We find that library developers often rephrase documents to improve their accuracies. For example, in J2SE 1.2.2, the document of the `java.util.GregorianCalendar` class includes a sentence: “Week 1 for a year is the first week that ...”, and the document is modified in J2SE 1.3.1: “Week 1 for a year is the earliest seven day period starting on `getFirstDayOfWeek()` that...”.

In many cases, a modified document has trivial revisions. While in other cases, revisions are non-trivial, and modified documents may indicate behavioral differences. For example, in ActiveMQ 5.3.0, the document of the `org.apache.activemq.broker.region.cursors.PendingMessageCursor.pageInList(int)` method is “Page in a restricted number of messages”. In ActiveMQ 5.3.1, the document is changed to “Page in a restricted number of messages and increment the reference count”. Based on the revision, the behavior of the method may change, and the changed method can increase the reference count. We further discuss this issue in Section 3.2.

Syntax 3.57% Library developers may produce a document with syntax errors, and fix them in a later version. For example, in J2SE 1.5, the document of the `java.util.ArrayList.remove(int)` method includes a sentence: “index the index of the element to removed”. Library developers fix this syntax error in J2SE 1.6, and the modified document is “index the index of the element to be removed”.

Typo 3.47% Library developers may produce some typos, and fix them in a later version (*e.g.*, `possible` → `possible`).

Format 0.84% Library developers may modify formats of some words for better presentation. For example, in J2SE 1.5, the document of the `java.util.Vector.setElementAt(E, int)` method includes a sentence: “the set method reverses ...”. In J2SE 1.6, library developers change the font of one word, and the modified document is “the set method reverses ...”. For better readability, a code element within a document is marked by the `@code` annotation in J2SE 1.6.

Implication 2: Many literal polishes such as those for fixing typos, syntax errors, and format issues can be avoided if researchers or practitioners propose an appropriate editor to library developers. It is challenging to implement such an editor for three reasons. (1) Many specialized terms and code names may be detected as typos. For example, although the Eclipse IDE can find some typos, it wrongly identifies “apple” as a typo

since it is a specialized term of computer science. (2) Code examples may be detected to include syntax errors. To check these code examples, an editor should understand corresponding programming languages. (3) Specific styles of API documentation may not be well supported by existing editors, and one such style is defined by the official guidance of Java documentation: “library developers should use ‘this’ instead of ‘the’ when referring to an object created from the current class”.

Finding 3: 22.62% of revisions are about programming tips as follows.

Notice 16.24% Library developers may add notices to describe API usages. Many notices start with the labeling word “Note”, but following this style is not a strict requirement. For example, in lucene 3.0.0, library developers add two sentences to the document of the `org.apache.lucene.util.CloseableThreadLocal` class without any labels: “We can not rely on `ThreadLocal.remove()`... You should not call `close` until all threads are done using the instance”. In some cases, notices even have no modal verbs such as *must* and *should*. For example, in J2SE 1.5, a notice is added to the document of the `java.util.Observable.deleteObserver(Observer)` method: “Passing `null` to this method will have no effect”.

Library developers may also modify a notice. For example, in J2SE 1.3.1, the document of the `java.util.AbstractCollection.clear()` method has a sentence: “Note that this implementation will throw an `UnsupportedOperationException` if the iterator returned by this collection’s iterator method does not implement the `remove` method”. In J2SE 1.4.2, the modified document includes another condition: “... not implement the `remove` method and this collection is non-empty”. Library developers may even delete notice. For example, in lucene 3.0.0, library developers delete a notice of the `org.apache.lucene.index.IndexWriter.getReader()` method: “You must close the `IndexReader` returned by this method once you are done using it”. It seems that programmers do not have to close the reader explicitly any more.

Code Example 4.36% Library developers may add code examples to illustrate API usages, and later fix defects in code examples. For example, in J2SE 1.5, the document of the `java.util.List.hashCode()` method has a code sample: “`hashCode = 1;...`”, and in J2SE 1.6, a defect is fixed: “`int hashCode = 1;...`”.

As pointed out by Kim *et al.* [12], API documentation in Java typically does not contain as many code examples as API documentation in other languages (*e.g.*, MSDN⁹). Still, library developers of Java libraries are reluctant to add code examples to API documents. Although code examples are useful to programmers, some library developers believe that API documentation should not contain code examples. In particular, the official guidance of Java documentation says “What separates API specifications from a programming guide are examples,...”, so adding many code examples to documentation is against the guidance.

New Description 2.02% Some API elements may not have any documents, or have only automatically generated documents without any true descriptions of usages. In some cases, an API element is found not straightforward to use, so library developers add new descriptions for the API element. For example, in J2SE 1.5, the document of the `java.util.ListResourceBundle.getContents()` method has only one

⁹ <http://msdn.microsoft.com/>

sentence: “See class description”. However, in J2SE 1.6, library developers add detailed explanations to the method: “Returns an array in which each item is a pair of objects in an `Object` array. The first element of each pair is the key, which must be a `String`, and the second element is the value associated with that key. See the class description for details”.

Implication 3: Dekel and Herbsleb [7] show that programming tips such as notices are quite valuable to programmers, but we find that programming tips are challenging to identify since Java does not provide any corresponding annotations. If such annotations are available, tools (*e.g.*, the one proposed by Dekel and Herbsleb [7]) may assist programmers more effectively. In addition, although many programmers complain that API documentation in Java lacks code examples, some library developers are still reluctant to add more code samples partially because doing so violates the principle of writing Java documentation. Some tools (*e.g.*, the tool proposed by Kim *et al.* [12]) may help bridge the gap between programmers and library developers.

Besides the preceding findings (Findings 1–3), other 2.25% revisions cannot be put into the preceding categories. Some of such revisions are still valuable. For example, in lucene 2.9.1, the document of the `org.apache.lucene.analysis.standard.StandardTokenizer` class includes a sentence that describes a fixed defect and its related bug report: “As of 2.4, Tokens incorrectly identified as acronyms are corrected (see [LUCENE-1608](#))”. Tools can use this clue to build relations between bug reports and API code automatically.

3.2 RQ2: To What Degree Do Such Revisions Indicate Behavioral Differences?

In this paper, we refer to differences in input/output values, functionalities, and call sequences between two versions of an API library as behavioral differences of API elements. Some behavioral differences can be reflected from revisions of API documentation. In this section, we analyze behavioral differences based on textual revisions of exceptions, parameters, returns, rephrasing, notices, and example code, since these revisions can often indicate behavioral differences as shown in Section 3.1. In total, we find that 18.44% revisions indicate behavioral differences. We classify found behavioral differences into three primary categories, and Figure 3 shows the results. The vertical axis shows the primary categories, and the horizontal axis shows their proportions.

Finding 4: 41.99% of behavioral differences are about exceptions as follows.

Addition 39.19% Library developers may re-implement API methods to throw new exceptions, and add exception documents for these API methods. For example, in J2SE 1.3, library developers re-implement the `java.util.ResourceBundle.getObject(String)` method to throw a new exception (`NullPointerException`), and add a corresponding document. In total, we find that `NullPointerException`, `ClassCastException`, and `IllegalArgumentException` are the top three added exceptions.

Modification 2.04% Library developers may change thrown exceptions of API methods, and modify corresponding documents. For example, in J2SE 1.3, the `Vector.addAll(Collection)` method throws `ArrayIndexOutOfBoundsException`. The thrown exception is changed to `NullPointerException` in J2SE 1.4, and its document is also modified.

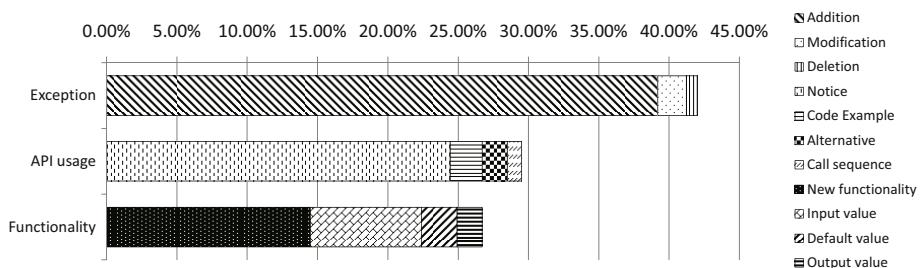


Fig. 3. Categories of behavioral differences

Deletion 0.76% Library developers may delete exceptions from API methods, and delete corresponding documents. For example, in ActiveMQ 5.1.0, the `org.apache.activemq.transport.tcp.SslTransportFactory.setKeyAndTrustManagers (KeyManager[], TrustManager[], SecureRandom)` method throws `KeyManagementException`. In ActiveMQ 5.2.0, library developers deprecate this method and delete its exception.

Implication 4: Library developers can add, modify, and delete exception documents, and these modifications indicate behavioral differences. In addition, we find that similar API methods often have similar revisions, and such a similarity could potentially be leveraged to better maintain API documentation.

Finding 5: 29.77% of behavioral differences are about API usage as follows.

Notice 24.68% As discussed in Finding 3, modifications of notices can reflect behavioral differences. For example, in J2SE 1.5, the document of the `java.util.Random.setSeed(long)` method has one notice: “Note: Although the seed value is an `AtomicLong`, this method must still be synchronized to ensure correct semantics of `haveNextNextGaussian`”. In J2SE 1.6, the notice is deleted. The revision indicates that the latter version does not have to be synchronized as the former version does.

Code example 2.29% Code examples explain API usages, so their revisions very likely reflect behavioral differences. For example, in lucene 2.9.0, the example code of keys in the `org.apache.lucene.search.Hits` class is as follow:

```
TopScoreDocCollector collector = new TopScoreDocCollector(hitsPerPage);
searcher.search(query, collector);
ScoreDoc[] hits = collector.topDocs().scoreDocs;
for (int i = 0; i < hits.length; i++) {
...
}
```

In lucene 2.9.1, the code example is modified as follows:

```
TopDocs topDocs = searcher.search(query, numHits);
ScoreDoc[] hits = topDocs.scoreDocs;
for (int i = 0; i < hits.length; i++) {
...
}
```

The version shows that programmers should follow a different way to attain a `ScoreDoc[]` object.

Alternative 1.78% When an API element becomes deprecated, library developers may refer to another API element for the deprecated one as alternatives. Revisions on alternatives very likely reflect behavioral differences. For example, the document of the

Table 2. Percentages of overall API differences

Library	1-2	2-3	3-4	4-5
J2SE	26.05%	59.04%	40.02%	22.60%
ActiveMQ	5.51%	4.95%	5.52%	1.64%
lucene	1.17%	0.51%	13.81%	0.55%
log4j	0.02%	0.00%	13.13%	5.47%
struct	28.17%	8.41%	4.60%	0.20%

deprecated `org.apache.lucene.analysis.StopAnalyzer.StopAnalyzer(Set)` constructor is “Use `StopAnalyzer(Set, boolean)` instead” in lucene 2.9.0. In lucene 2.9.1, library developers change the document to “Use `StopAnalyzer(Version, Set)` instead”. The revision indicates that another API method should be used to replace the deprecated API method.

Call sequence 1.02% The revisions of API call sequences very likely reflect behavioral differences. For example, in lucene 2.9.2, the document of the `org.apache.lucene.search.Scorer.score()` method has a sentence: “Initially invalid, until `DocIdSetIterator.next()` or `DocIdSetIterator.skipTo(int)` is called the first time”. In lucene 3.0.0, library developers modify this sentence: “Initially invalid, until `DocIdSetIterator.nextDoc()` or `DocIdSetIterator.advance(int)` is called the first time”. The revision indicates that programmers should call different API methods in the latter version from those in the former version.

Implication 5: As API usages are quite important to programmers, library developers take much effort to improve related documents. However, we find that Java has quite limited annotations to support API-usage documents. If such annotations are available, library developers can improve readability of API documentation, and programmers can better understand API evolution.

Finding 6: 26.71% of behavioral differences are about functionalities as follows.

New functionality 14.50% Library developers may implement new functionalities for some API elements, and revise corresponding documents. For example, `generic` is a new functionality introduced in J2SE 1.5, and many related documents are modified (*e.g.*, the document of the `java.util.Collections` class).

Input value 7.89% Library developers may change input ranges of some API methods, and revise corresponding documents. For example, in J2SE 1.3.1, the document of the `java.util.Properties.store(OutputStream, String)` method includes “The ASCII characters \, tab, newline, and carriage return are written as \\, \\t, \\n, and \\r, respectively”. In J2SE 1.4.2, library developers add a new ASCII translation for “form feed”, and the sentence is modified to “The ASCII characters \, tab, form feed, newline, and carriage return are written as \\, \\t, \\f \\n, and \\r, respectively”. The revision indicates that the latter version can accept more characters (*e.g.*, form feed) as inputs than the former version does.

Default value 2.54% Library developers may change default values of some API methods, and revise corresponding documents. For example, in J2SE 1.3.1, the document of the `java.util.Hashtable(Map t)` method informs that “The hashtable is

created with a capacity of twice the number of entries in the given Map or 11 (whichever is greater)". In J2SE 1.4.2, library developers delete the words about default values, and change the sentence to "The hashtable is created with an initial capacity sufficient to hold the mappings in the given Map". The revision indicates that the capacity of the latter version is different from the former version.

Output value 1.78% Library developers may change output values of some API methods, and revise corresponding documents. In some cases, library developers may find that some output values are not straightforward, so they add documents to these output values. For example, in log4j 1.2.15, the document of the `org.apache.log4j.Appender.getName()` method does not have any sentences about return values. In log4j 1.2.16, library developers add one sentence about return values: "return name, may be null". The sentence explains that the return value can be `null`. The revision indicates that the latter version can return `null` values, whereas the former version may not. Still, it is difficult to fully determine whether such a revision indicates behavioral differences or not, and we further discuss the issue in Section 4.

Implication 6: Besides adding new functionalities, library developers also change functionalities by modifying default values and input/output values of some API methods. Although some annotations (*e.g.*, `@param` and `@return`) are provided to describe input and output parameters, value ranges are difficult to identify since they are usually mixed with names and descriptions of parameters. In addition, other revisions (*e.g.*, revisions of default values) even have no corresponding annotations. If such annotations are supported, it may be easier to analyze those revisions for behavioral differences.

Besides the preceding findings (Findings 4–6), other 1.53% revisions of behavioral differences cannot be put into the preceding categories.

3.3 RQ3: How Frequently Are API Elements and Their Documentation Changed?

Sections 3.1 and 3.2 have investigated detailed revisions in API documentation. In this section, we present an overall evolution frequency of API differences over versions of all the libraries listed in Table 1. Given two library versions L_1 and L_2 , we define the API difference between the two versions of the API library as follows:

$$Dif(L_1, L_2) = \frac{\text{additions} + \text{removals} + 2 \times \text{modifications}}{\text{sum of public elements in } L_1 \text{ and } L_2} \quad (1)$$

In Equation 1, additions denote newly added API elements; removals denote deleted API elements; and modifications denote API elements whose declarations or documents are modified. We count the number of modifications twice, since each modification can be considered as a removal and an addition.

Finding 7: Contrary to normal expectations, API libraries between two nearby versions are typically quite different. The API differences between two library versions are largely proportional to the differences between the version numbers of the two versions, and *additions* and *modifications* account for most of the proportions by evolution types.

Table 3. Percentages of API differences by types

(1) J2SE				(2) ActiveMQ			
V	A	R	M	V	A	R	M
1-2	10.37%	0.61%	89.02%	1-2	56.80%	5.78%	37.42%
2-3	15.02%	0.83%	84.15%	2-3	67.60%	2.23%	30.17%
3-4	20.12%	4.55%	75.33%	3-4	42.84%	4.91%	52.25%
4-5	12.68%	1.40%	85.92%	4-5	41.61%	7.42%	50.97%

(3) lucene				(4) log4j				(5) struct			
V	A	R	M	V	A	R	M	V	A	R	M
1-2	22.89%	1.21%	75.90%	1-2	50.00%	0.00%	50.00%	1-2	26.88%	39.13%	33.99%
2-3	7.89%	0.00%	92.11%	2-3	n/a	n/a	n/a	2-3	39.13%	12.32%	48.55%
3-4	8.71%	41.41%	49.88%	3-4	42.11%	6.57%	51.32%	3-4	52.05%	11.06%	36.89%
4-5	14.71%	2.94%	82.35%	4-5	38.74%	0.00%	61.26%	4-5	0.00%	0.00%	100.00%

Table 2 shows the overall API differences. Each column denotes the API difference between the two versions of a library. For example, Column “1-2” lists the API differences between a v1 version and a v2 version. From Table 2, we find that two versions of a library typically provide different API elements, and only two versions of log4j have exactly the same API elements. For each library, API differences are largely proportional to differences of version numbers. For example, API differences of ActiveMQ are proportional to the differences of its version numbers shown in Table 1.

Table 3 shows the proportions of evolution types. For each library, Column “V” lists versions. For example, Row “1-2” shows the proportions between a v1 version and a v2 version. Column “A” denotes proportions of *additions*. Column “R” denotes proportions of *removals*. Column “M” denotes proportions of *modifications*. From the results of Table 3, we find that *additions* and *modifications* account for the most of the proportions of evolution types. It seems that library developers are often reluctant to remove API elements, possibly for the consideration of compatibility across versions.

Implication 7: Based on our results, the API differences between two versions of an API library increase with the differences between their version numbers, so analysis tools for API evolution should deal with more API differences between versions with more different version numbers. Modifications account for the largest proportions. As modifications keep signatures of API methods unchanged, they typically do not cause compilation errors. Thus, analysis tools need to identify and deal with modifications carefully to ensure that the process of API evolution does not introduce new defects into client code.

3.4 Summary

Overall, we find that API documentation between two nearby versions can be quite different, and API differences between versions are proportional to differences of version numbers (Finding 7). Our findings are valuable to better understand API documentation evolution by highlighting the following aspects:

Evolution distribution. Most revisions occur in annotations, but some of annotations are difficult to maintain (Finding 1). Literal polishes account for the second place, and about 30% effort can be saved when an appropriate editor is available (Finding 2). Programming tips account for the third place, and documents on programming tips are challenging to identify since there are no corresponding annotations (Finding 3).

Behavioral differences. Most behavioral differences occur in revisions of exceptions (Finding 4). Although various revisions can indicate behavioral differences, no corresponding annotations exist to support these revisions (Findings 5 and 6).

3.5 Threats to Validity

The threats to external validity include the representativeness of the subjects in true practice. Although we choose five widely used real-world libraries as subjects, our empirical study investigated limited libraries with limited versions, so some findings (*e.g.*, percentages) may not be general. This threat could be reduced by investigating more versions of more libraries. The threats to internal validity include the human factors within our methodology. Although we tried our best to reduce the subjectivity by using double verification, to further reduce this threat, we need to invite more participants to verify our results.

4 Discussion and Future Work

In this section, we discuss issues and our future work.

Variance across version changes. As shown in Table 2, percentages of changes between versions are not fully uniform with variances. To investigate such variances, we plan to use finer-grained analysis in future work. In particular, we plan to investigate the distribution of those variances, their associations, and their styles of common changes for better understanding API evolution.

Determining behavioral differences. It is challenging to automatically determine behavioral differences through only documentation analysis or only code analysis (*e.g.*, code refactoring typically does not cause any behavioral differences). In future work, we plan to combine documentation analysis with code analysis to better determine behavioral differences than with individual techniques.

Benefits of our findings. Our findings are beneficial to programmers, library developers, IDE developers, and researchers. For example, for IDE developers, as our findings reveal that many revisions (*e.g.*, revisions of version numbers) are of little interest, it can be ineffective if IDE developers design an IDE where library developers are required to manually make all types of revisions of API documentation. As another example, for researchers, Mariani *et al.* [14] can also improve their approach that identifies anomalous events, if they consider modified notices and code examples that may lead to anomalous events. Furthermore, we released our results on our project website, so others can analyze benefits of our findings under their contexts.

5 Related Work

Our quantitative study is related to previous work as follows.

Natural language analysis in software engineering. Researchers have proposed approaches to analyze natural language documents in software engineering. Tan *et al.* [21] proposed an approach to infer rules and to detect defects from single sentences of comments. Zhong *et al.* [26] proposed an approach that infers resource specifications from descriptions of multiple methods. Tan *et al.* [22] proposed an approach that infers annotations from both code and comments to detect concurrency defects. Dekel and Herb-sleb [7] proposed an approach that pushes rule-containing documents to programmers. Horie and Chiba [11] proposed an extended Javadoc tool that provides new tags to maintain crosscutting concerns in documentation. Buse and Weimer [4, 3] presented various automatic techniques for exception documentation and synthesizing documentation for arbitrary programme differences across versions. Sridhara *et al.* [20] proposed an approach that infers comments of Java methods from API code. Würsch *et al.* [23] proposed an approach that supports programmers with natural language queries. Kof [13] used POS tagging to identify missing objects and actions in requirement documents. Instead of proposing a new approach, we conducted an empirical study that motivates future work on analyzing API documentation in natural languages.

API translation. Researchers have proposed approaches to translate APIs from one API library to another. Henkel and Diwan [10] proposed an approach that captures and replays API refactoring actions to update the client code. Xing and Stroulia [24] proposed an approach that recognizes the changes of APIs by comparing the differences between two versions of libraries. Balaban *et al.* [2] proposed an approach to migrate client code when mapping relations of libraries are available. Dagenais and Robillard [5] proposed an approach that recommends relevant changes of API elements based on comparing API code. Zhong *et al.* [25] proposed an approach that mines API mapping relations for translating APIs in one language to another. Our empirical study reveals various findings and implications on API documentation evolution, and these findings are valuable to improve exiting API translation approaches.

Empirical studies on software evolution or API libraries. Researchers have conducted various empirical studies on software evolution or API libraries. Ruffell and Selby's empirical study [19] reveals that global data is inherent and follows a wave pattern during software evolution. Geiger *et al.*'s empirical study [9] reveals that the relation between code clones and change couplings is statistically unverifiable, although they find many such cases. Bacchelli *et al.*'s empirical study [1] reveals that the discussions of an artifact in email archives and the defects of the artifact are significantly correlated. Novick and Ward's empirical study [16] reveals that many programmers are reluctant to seek help from documentation. Robillard and DeLine [18] conducted an empirical study to understand obstacles to learn APIs, and present many implications to improve API documentation. Padoleau *et al.* [17] presented an empirical study on taxonomies of comments in operating system code. Dagenais and Robillard [6] conducted a qualitative study on creation and evolution of documentation, whereas we conducted a quantitative study on the evolution. Dig and Johnson's empirical studies [8] reveal that

refactoring plays an important role in API evolution, and some breaking changes may cause behavioral differences or compilation errors in client code. Our empirical study focuses on the evolution of API documentation, complementing their studies.

6 Conclusion

A quantitative study on API documentation evolution is quite valuable for both programmers and library developers to better understand evolution, and it is difficult to conduct such a study due to various challenges. In this paper, we present an analysis methodology to analyze the evolution of API documentation. We conduct a quantitative study on API documentation evolution of five real-world Java libraries. The results show that API documentation undergoes frequent evolution. Understanding these results helps programmers better learn API documentation evolution, and guides library developers better in maintaining their documentation.

Acknowledgments

We appreciate anonymous reviewers for their supportive and constructive comments. The authors from Chinese Academy of Sciences are sponsored by the National Basic Research Program of China (973) No. 2007CB310802, the Hi-Tech Research and Development Plan of China (863) No. 2007AA010303, the National Natural Science Foundation of China No. 90718042, 60803023, 60873072, and 60903050, and the CAS Innovation Program ISCAS2009-GR. Tao Xie's work is supported in part by NSF grants CNS-0716579, CCF-0725190, CCF-0845272, CCF-0915400, CNS-0958235, an NCSU CACC grant, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU SOSI.

References

- [1] Bacchelli, A., Ambros, M.D., Lanza, M.: Are popular classes more defect prone? In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 59–73. Springer, Heidelberg (2010)
- [2] Balaban, I., Tip, F., Fuhrer, R.: Refactoring support for class library migration. In: Proc. 20th OOPSLA, pp. 265–279 (2005)
- [3] Buse, R., Weimer, W.: Automatic documentation inference for exceptions. In: Proc. ISSTA, pp. 273–282 (2008)
- [4] Buse, R., Weimer, W.: Automatically documenting program changes. In: Proc. 26th ASE, pp. 33–42 (2010)
- [5] Dagenais, B., Robillard, M.: Recommending adaptive changes for framework evolution. In: Proc. 30th ICSE, pp. 481–490 (2009)
- [6] Dagenais, B., Robillard, M.P.: Creating and evolving developer documentation: Understanding the decisions of open source contributors. In: Proc. 18th ESEC/FSE, pp. 127–136 (2010)
- [7] Dekel, U., Herbsleb, J.D.: Improving API documentation usability with knowledge pushing. In: Proc. 31st ICSE, pp. 320–330 (2009)

- [8] Dig, D., Johnson, R.: How do APIs evolve? a story of refactoring. *Journal of software maintenance and evolution: Research and Practice* 18(2), 83–107 (2006)
- [9] Geiger, R., Fluri, B., Gall, H., Pinzger, M.: Relation of code clones and change couplings. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 411–425. Springer, Heidelberg (2006)
- [10] Henkel, J., Diwan, A.: CatchUp!: capturing and replaying refactorings to support API evolution. In: Proc. 27th ICSE, pp. 274–283 (2005)
- [11] Horie, M., Chiba, S.: Tool support for crosscutting concerns of API documentation. In: Proc. 8th AOSD, pp. 97–108 (2010)
- [12] Kim, J., Lee, S., Hwang, S., Kim, S.: Adding examples into Java documents. In: Proc. 24th ASE, pp. 540–544 (2009)
- [13] Kof, L.: Scenarios: Identifying missing objects and actions by means of computational linguistics. In: Proc. 15th RE, pp. 121–130 (2007)
- [14] Mariani, L., Pastore, F., Pezze, M.: A toolset for automated failure analysis. In: Proc. 31st ICSE, pp. 563–566 (2009)
- [15] Mens, T., Tourwe, T.: A survey of software refactoring. *IEEE Transactions on software engineering* 30(2), 126–139 (2004)
- [16] Novick, D.G., Ward, K.: Why don't people read the manual? In: Proc. 24th SIGDOC, pp. 11–18 (2006)
- [17] Padolieau, Y., Tan, L., Zhou, Y.: Listening to programmers Taxonomies and characteristics of comments in operating system code. In: Proc. 31st ICSE, pp. 331–341 (2009)
- [18] Robillard, M.P., DeLine, R.: A field study of API learning obstacles. *Empirical Software Engineering* (to appear, 2011)
- [19] Ruffell, F., Selby, J.: The pervasiveness of global data in evolving software systems. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 396–410. Springer, Heidelberg (2006)
- [20] Sridhara, G., Hill, E., Muppaneni, D., Pollock, L.L., Vijay-Shanker, K.: Towards automatically generating summary comments for Java methods. In: Proc. 25th ASE, pp. 43–52 (2010)
- [21] Tan, L., Yuan, D., Krishna, G., Zhou, Y.: iComment: Bugs or bad comments. In: Proc. 21st SOSP, pp. 145–158 (2007)
- [22] Tan, L., Zhou, Y., Padolieau, Y.: aComment: Mining annotations from comments and code to detect interrupt-related concurrency bugs. In: Proc. 33rd ICSE (to appear, 2011)
- [23] Würsch, M., Ghezzi, G., Reif, G., Gall, H.C.: Supporting developers with natural language queries. In: Proc. 32nd ICSE, pp. 165–174 (2010)
- [24] Xing, Z., Stroulia, E.: API-evolution support with Diff-CatchUp. *IEEE Transactions on Software Engineering* 33(12), 818–836 (2007)
- [25] Zhong, H., Thummalapenta, S., Xie, T., Zhang, L., Wang, Q.: Mining API mapping for language migration. In: Proc. 32nd ICSE, pp. 195–204 (2010)
- [26] Zhong, H., Zhang, L., Xie, T., Mei, H.: Inferring resource specifications from natural language API documentation. In: Proc. 24th ASE, pp. 307–318 (2009)

An Empirical Study of Long-Lived Code Clones

Dongxiang Cai¹ and Miryung Kim²

¹ Hong Kong University of Science and Technology

`caidx@cse.ust.hk`*

² The University of Texas at Austin

`miryung@ece.utexas.edu`

Abstract. Previous research has shown that refactoring code clones as soon as they are formed or discovered is not always feasible or worthwhile to perform, since some clones never change during evolution and some disappear in a short amount of time, while some undergo repetitive similar edits over their long lifetime.

Toward a long-term goal of developing a recommendation system that selectively identifies clones to refactor, as a first step, we conducted an empirical investigation into the characteristics of long-lived clones. Our study of 13558 clone genealogies from 7 large open source projects, over the history of 33.25 years in total, found surprising results. The size of a clone, the number of clones in the same group, and the method-level distribution of clones are not strongly correlated with the survival time of clones. However, the number of developers who modified clones and the time since the last addition or removal of a clone to its group are highly correlated with the survival time of clones. This result indicates that the evolutionary characteristics of clones may be a better indicator for refactoring needs than static or spatial characteristics such as LOC, the number of clones in the same group, or the dispersion of clones in a system.

Keywords: Software evolution, empirical study, code clones, refactoring.

1 Introduction

Code clones are code fragments similar to one another in syntax and semantics. Existing research on code cloning indicates that a significantly large portion of software (e.g. gcc-8.7% [9], JDK-29% [14], Linux-22.7% [27], etc.) contains code duplicates created by copy and paste programming practices. Though code cloning helps developers to reuse existing design and implementation, it could incur a significant maintenance cost because programmers need to apply repetitive edits when the common logic among clones changes. Neglecting to update clones consistently may introduce a bug.

* This research was conducted while the first author was a graduate student intern at The University of Texas at Austin.

Refactoring is defined as a disciplined technique for restructuring existing software systems, altering a program’s internal structure without changing its external behavior [10]. Because refactoring is considered a key to keeping source code easier to understand, modify, and extend, previous research effort has focused on automatically identifying clones [15,4,2,21,13].

However, recent studies on code clones [7,18,19,20,26] indicated that cloning is not necessarily harmful and that refactoring may not be always applicable to or beneficial for clones. In particular, our previous study of clone evolution [20] found that (1) some clones never change during evolution, (2) some clones disappear after staying in a system for only a short amount of time due to divergent changes, and (3) some clones stay in a system for a long time and undergo consistent updates repetitively, indicating a high potential return for the refactoring investment. These findings imply that it is crucial to *selectively identify* clones to refactor.

We hypothesize that the benefit of clone removal may depend on how long clones survive in the system and how often they require similar edits over their lifetime. Toward a long-term goal of developing a system that recommends clones to refactor, as a first step, we conducted an empirical investigation into the characteristics of long-surviving clones. Based on our prior work on clone genealogies—an automatically extracted history of clone evolution from a sequence of program versions [20]—we first studied various factors that may influence a clone’s survival time, such as the number of clones in the same group, the number of consistent updates to clones in the past, the degree of clone dispersion in a system, etc. In total, we extracted 34 attributes from a clone genealogy and investigated correlation between each attribute and a clone’s survival time in terms of the number of days before disappearance from a system. Our study found several surprising results. The more developers maintain code clones, the longer the clones survive in a system. The longer it has been since the time of the last addition or deletion of a clone to its clone group, the longer the clones survive in a system. On the other hand, a clone’s survival time did not have much correlation with the size of clones, the number of clones in the same group, and the number of methods that the clones are located in.

For each subject, we developed a decision-tree based model that predicts a clone survival time based on its characteristics. The model’s precision ranges from 58.1% to 79.4% and the recall measure ranges from 58.8% to 79.3%. This result shows promise in developing a refactoring recommendation system that selects long-lived clones.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 gives background of our previous clone genealogy research and Section 4 describes the characteristics of clone genealogy data and the subject programs that we studied. Section 5 describes correlation analysis results and Section 6 presents construction and evaluation of decision tree-based prediction models. Section 7 discusses threats to validity, and Section 8 summarizes our contributions.

2 Related Work

This section describes tool support for identifying refactoring opportunities, empirical studies of code cloning, and clone evolution analysis.

Identification of Refactoring Opportunities. Higo et al. [13] propose *Aries* to identify refactoring candidates based on the number of assigned variables, the number of referred variables, and clone dispersion in the class hierarchy. *Aries* suggests two types of refactorings, *extract method* and *pull-up method* [10]. A refactoring can be suggested if the clone metrics satisfy certain predefined values. Komondoor's technique [22] extracts non-contiguous lines of clones into a procedure that can then be refactored by applying an *extract method* refactoring. Koni-N'Sapu [23] provides refactoring suggestions based on the location of clones with respect to a system's class hierarchy. Balazinska et al. [2] suggest clone refactoring opportunities based on the differences between the cloned methods and the context of attributes, methods, and classes containing clones. *Breakaway* [8] automatically identifies detailed structural correspondences between two abstract syntax trees to help programmers generalize two pieces of similar code. Several techniques [35,34,33,11,28] automatically identify bad-smells that indicate refactoring needs. For example, Tsantalis and Chatzigeorgiou's technique identifies *extract method* refactoring opportunities using static slicing. Our work is different from these refactoring opportunity identification techniques in that it uses clone *evolution history* to predict how long clones are likely to survive in a system.

Studies about Cloning Practice. Cordy [7] notes that cloning is a common method of risk minimization used by financial institutions because modifying an abstraction can introduce the risk of breaking existing code. Fixing a shared abstraction is costly and time consuming as it requires any dependent code to be extensively tested. On the other hand, clones increase the degrees of freedom in maintaining each new application or module. Cordy noted that propagating bug fixes to clones is not always a desired practice because the risk of changing an already properly working module is too high.

Godfrey et al. [12] conducted a preliminary investigation of cloning in Linux SCSI drivers and found that super-linear growth in Linux is largely caused by cloning of drivers. Kapser and Godfrey [18] further studied cloning practices in several open source projects and found that clones are not necessarily harmful. Developers create new features by starting from existing similar ones, as this cloning practice permits the use of stable, already tested code. While interviewing and surveying developers about how they develop software, LaToza et al. [26] uncovered six patterns of why programmers create clones: repeated work, example, scattering, fork, branch, and language. For each pattern, less than half of the developers interviewed thought that the cloning pattern was a problem. LaToza et al.'s study confirms that most cloning is unlikely to be created with ill intentions. Rajapakse et al. [30] found that reducing duplication in a web application had negative effects on the extensibility of an application. After significantly reducing the size of the source code, a single change often required testing a vastly

larger portion of the system. Avoiding clones during initial development could contribute to a significant overhead. These studies indicate that not all clones are harmful and it is important to *selectively identify clones to refactor*.

Clone Evolution Analysis. While our study uses the evolutionary characteristics captured by the clone genealogy model [20], the following clone evolution analyses could serve as a basis for generating clone evolution data. The evolution of code clones was analyzed for the first time by Lagu   et al. [25]. Aversano et al. [1] refined our clone genealogy model [20] by further categorizing the *Inconsistent Change* pattern into the *Independent Evolution* pattern and the *Late Propagation* pattern. Krinke [24] also extended our clone genealogy analysis and independently studied clone evolution patterns. Balint et al. [3] developed a visualization tool to show who created and modified code clones, the time of the modifications, the location of clones in the system, and the size of code clones.

Classification of Code Clones. Bellon et al. categorized clones into Type 1 (an exact copy without modifications), Type 2 (a syntactically identical copy) and Type 3 (a copy with further modifications, e.g., addition and deletion of statements) in order to distinguish the kinds of clones that can be detected by existing clone detectors [5]. Kapser and Godfrey [17,16] taxonomized clones to increase the user comprehension of code duplication and to filter false positives in clone detection results. Several attributes of a clone genealogy in Section 4 are motivated by Kapser and Godfrey's region and location based clone filtering criteria. Our work is different from these projects by identifying the characteristics of long-lived clones.

3 Background on Clone Genealogy and Data Set

A clone genealogy describes how groups of code clones change over multiple versions of the program. A clone group is a set of clones considered *equivalent* according to a clone detector. For example, clone *A* and *B* belong to the same group in version *i* because a clone detector finds them equivalent. In a clone's genealogy, a group to which the clone belongs is traced to its origin clone group in the previous version. The model associates related clone groups that have originated from the same ancestor group. In addition, the genealogy contains information about how each element in a group of clones changed with respect to other elements in the same group. The detail description on the clone genealogy representation is described elsewhere [20]. The following evolution patterns describe all possible changes in a clone group.

Same: all code snippets in the new version's clone group did not change from the old version's clone group.

Add: at least one code snippet is newly added to the clone group.

Subtract: at least one code snippet in the old version's clone group does not appear in the corresponding clone group in the new version.

Consistent Change: all code snippets in the old version's clone group have changed consistently; thus, they all belong to the new clone group.

Inconsistent Change: at least one code snippet in the old version's clone group changed inconsistently; thus, it no longer belongs to the same group in the new version.

Shift: at least one code snippet in the new clone group partially overlaps with at least one code snippet in the original clone group.

A *clone lineage* is a directed acyclic graph that describes the evolution history of a sink node (clone group). A clone group in the k^{th} version is connected to a clone group in the $k - 1^{\text{th}}$ version by an evolution pattern. For example, Figure 1 shows a clone lineage following the sequence of **Add**, **Same**, **Consistent Change**, **Consistent Change**, and **Inconsistent Change**. In the figure, code snippets with the similar content are filled with the same shade.

A *clone genealogy* is a set of clone lineages that have originated from the same clone group. A clone genealogy is a connected component where every clone group is connected by at least one evolution pattern. A clone genealogy approximates how programmers create, propagate, and evolve code clones.

Clone genealogies are classified into two groups: *dead* genealogies that do not include clone groups of the final version and *alive* genealogies that include clone groups of the final version. We differentiate a dead genealogy from an alive genealogy because only dead genealogies provide information about how long clones stayed in the system before they disappeared. On the other hand, for an alive genealogy, we cannot tell how long its clones will survive because they are still evolving. In Figure 1, G_1 is a dead genealogy with the age 5, and G_2 is an alive genealogy with the age 4. Dead genealogies are essentially genealogies that disappeared because clones were either refactored, because they were deleted by a programmer, or because they are no longer considered as clones by a clone detector due to divergent changes to the clones.

To extract clone evolution histories with respect to this model, our clone genealogy extractor (CGE) takes a sequence of program versions as input and

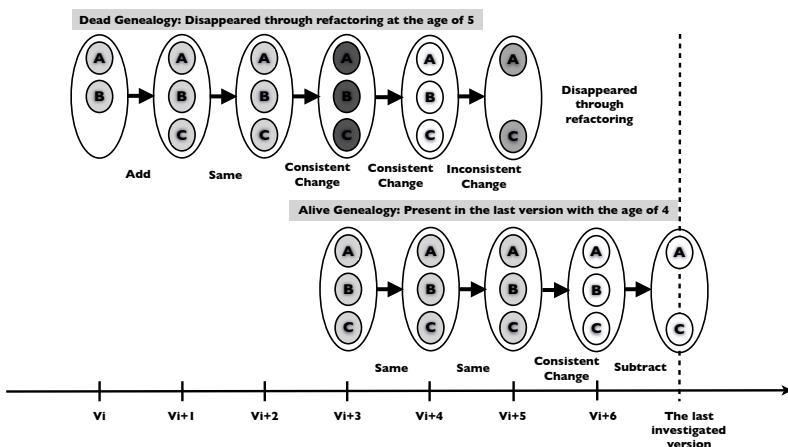


Fig. 1. Example clone genealogies: G_1 (above) and G_2 (below)

uses CCFinder [15] to find clones in each version. It then relates clones between consecutive versions based on a textual similarity computed by CCFinder and a location overlapping score computed by *diff*, a line-level program differencing tool. Basically, if the similarity between a clone of version i and a clone of a version $i + 1$ is greater than a similarity threshold, sim_{th} , their container clone groups are mapped.

Data sets. For our study, we extracted clone genealogy data from seven projects: Eclipse JDT core, jEdit, HtmlUnit, JFreeChart, Hadoop.common, Hadoop.pig, and Columba. Table 1 summarizes the size of subject programs and the number of studied versions. We constructed genealogies at a temporal granularity of releases instead of check-ins because our goal is not to understand fine-grained evolution patterns but to correlate clones' characteristics with a survival time. In total, we studied 7 large projects of over 100KLOC, in total 33.25 years of release history. Table 2 summarizes the number of dead and alive genealogies. In our analysis, we removed dead genealogies of age 0, because they do not provide much meaningful information about evolutionary characteristics.

Table 1. Description of Java subject programs

project	URL	LOC	duration	# of check-ins	# of versions
Columba	http://sourceforge.net/projects/columba	80448 ~ 194031	42 months	420	420
Eclipse	http://www.eclipse.org	216813 ~ 424210	92 months	13790	21
common	http://hadoop.apache.org/common	226643 ~ 315586	14 months	410	18
pig	http://hadoop.apache.org/pig	46949 ~ 302316	33 months	906	8
HtmlUnit	http://htmlunit.sourceforge.net	35248 ~ 279982	94 months	5850	22
jEdit	http://www.jedit.org	84318 ~ 174767	91 months	3537	26
JFreeChart	http://www.jfree.org/jfreechart	284269 ~ 316954	33 months	916	7

Table 2. Clone genealogies ($min_{token}=40$, $sim_{th} = 0.8$)

# of genealogies	Columba	Eclipse	common	pig	HtmlUnit	jEdit	JFreeChart
Total	556	3190	3094	3302	1029	654	1733
Alive genealogies	452	1257	627	2474	500	232	1495
Dead genealogies	104	1933	2467	828	529	422	238
# of dead genealogies with age>0	102	1826	455	422	425	245	219

4 Encoding Clone Genealogy Characteristics

To study the characteristics of long-lived clones, we encode a clone genealogy into a feature vector, which consists of a set of attributes. When measuring spatial characteristics of clones, we use information from the last version of a genealogy. For example, to measure the average size of clones in Genealogy $G1$ in Figure 1, we use the average size of clone A and C in the genealogy's last version, V_{i+5} . This section introduces each attribute and the rationale of choosing the attribute.

The number of clones in each group and the average size of a clone. The more clones exist in each clone group and the larger the size of each clone, it

may require more effort for a developer to remove those clones, contributing to a longer survival time.

a_0 : the total LOC (lines of code) of clones in a genealogy

a_1 : the number of clones in each group

a_2 : the average size of a clone in terms of LOC

Addition. Addition of new clones to a genealogy could imply that the clones are still volatile, or that it is becoming hard to maintain the system without introducing new clones, indicating potential refactoring needs.

a_3 : the number of Add evolution patterns

a_4 : the relative timing of the last Add pattern with respect to the age of a genealogy

Consistent update. If clones require similar edits repetitively over their lifetime, removal of those clones could provide higher maintenance cost-savings than removing unchanged clones.

a_5 : the number of Consistent Change patterns

a_6 : the relative timing of the last Consistent Change pattern with respect to the age of a genealogy

Subtraction. A Subtract pattern may indicate a programmer removed only a subset of existing clones.

a_7 : the number of Subtract patterns

a_8 : the relative timing of the last Subtract pattern with respect to the age of a genealogy

Inconsistent update. An Inconsistent Change pattern may indicate that the programmer forgot to update clones consistently, and thus a programmer may prefer to refactor such clones early to prevent inconsistent updates in the future.

a_9 : the number of Inconsistent Change patterns

a_{10} : the relative timing of the last Inconsistent Change pattern with respect to the age of a genealogy

File modification. If a file containing clones was modified frequently, it may indicate that those clones are likely to be removed early.

a_{11} : the number of times that files containing clones were modified.

Developers. The more developers are involved in maintaining clones, it may be harder to refactor the clones.

a_{12} : the number of developers involved in maintaining clones.

a_{13} : the distribution of file modifications in terms of developers.

If the following entropy measure—a well-known measure of uncertainty [31]—is low, that means only a few developers make most of the modifications. If the entropy is high, all developers equally contribute to the modifications. The entropy measure is defined as follows: $entropy = \sum_{i=1}^n -plog(p_i)$, where p_i is the probability of a file modification belonging to author i , when n unique authors maintain the file.

Dispersion. The farther clones are located from one another, the harder it is to find and refactor them. Inspired by Kapser and Godfrey's clone taxonomy [17],

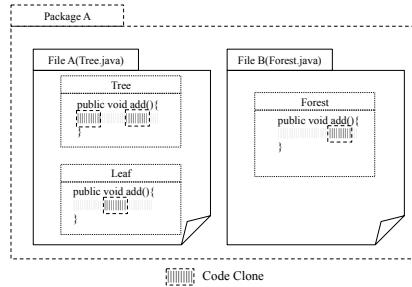


Fig. 2. Example physical distribution of code clones

we count the number of unique methods, classes, files, packages, and directories the clones are located. Table 3 shows that most clones are located within the same class or package, only 4.0% to 29.5% of clones are located within the same method, and only 2.8% to 31.1% clones are scattered across different directories. We also used an entropy measure to characterize the physical distribution of clones at a different level (method, class, file, package, and directory respectively) by defining p_i to be the probability of clones located in location i . For example, in Figure 2, the dispersion entropy at a method level is 1.5, the entropy at a file level is 0.81, and the entropy at a package level is 0. If the entropy is low, clones are concentrated in only a few locations. If the entropy is high, clones are equally dispersed across different locations.

a_{14} : Dispersion of clones into five nominal labels: 'within the same method', 'within the same class', 'within the same file', 'within the same package', and 'across multiple directories.'

a_{15} : The number of unique methods that clones in the *last* version are located.

a_{16} : The distribution of clones in the *last* version at a method level according to the entropy measure.

Table 3. Characteristics of studied clones

		Columba	Eclipse	common	pig	HtmlUnit	jEdit	JFreeChart
Age	Average age (days)	538.6	435.1	72.49	136.6	292.8	640.9	229.0
	Min	1.1	68.7	34.0	30.0	6.9	13.3	11.1
	Max	1222.2	2010.0	585.0	536.9	2122.4	2281.7	415.0
# of clones in each group	Average	3.38	3.37	3.67	4.54	4.43	4.10	3.26
	Min	2	2	2	2	2	2	2
	Max	21	112	53	115	62	120	42
Size (LOC)	Average clone size	12.97	18.59	16.22	14.38	14.34	12.10	15.77
	Min	2	3	3	3	3	4	4
	Max	38.5	343.4	79	70	92	60	75
Dispersion	% clones in the same method	27.5%	29.5%	13.4%	12.8%	4%	24.5%	22.8%
	% clones in the same class	61.8%	60.6%	32.5%	49.1%	44.9%	75.9%	31.5%
	% clones in the same file	61.8%	61.0%	48.8%	49.5%	45.2%	75.9%	31.5%
	% clones in the same package	80.4%	83.5%	71.4%	94.1%	87.3%	90.6%	58.9%
	% clones in the same directory	87.3%	83.9%	75.4%	97.2%	88.5%	93.9%	68.9%

a_{17} : The number of unique methods that clones in *all* versions are located.

a_{18} : The distribution of clones in *all* versions at a method level according to the entropy measure.

We then created similar attributes at the level of class (a_{19} to a_{22}), file (a_{23} to a_{26}), package (a_{27} to a_{30}), and directory (a_{31} to a_{34}) by replicating attributes (a_{15} to a_{18}).

Clone survival time (class label). The last attribute a_{35} is the age of a genealogy in terms of the number of days.

a_{35} : the age of a genealogy in terms of the number of days

In the next section, we conduct a correlation analysis between each of the 34 attributes (a_1 to a_{34}) with a clone survival time (a_{35}).

5 Characteristics of Long-Lived Clones

To understand the characteristics of long-lived clones, we measured Pearson's correlation coefficient between each attribute and a clone genealogy survival time [32]. In our analysis, we used only dead genealogies, because alive genealogies are still evolving and thus cannot be used to predict how long clones would survive before they disappear. Table 4 shows the result of top 5 and bottom 5 attributes in terms of correlation strength.

The result indicates the more developers maintained clones, the longer the survival time of a clone genealogy (a_{12}). The more uniformly developers contribute to maintaining clones, the longer time it takes for the clones to be removed (a_{13}). The longer it has been since the last addition or deletion of a clone, the longer it takes for them to be removed (a_4 and a_8). On the other hand, the size of clones (LOC), the number of clones in each group, and the physical dispersion of clones do not affect a clone survival time much (a_0 , a_1 , a_{31} , and a_{32}). This implies that the size and the number of clones do not play much role in estimating a clone survival time; however, it may be harder to remove those clones changed by a large number of developers.

For example, a clone genealogy id 1317 from Eclipse JDT disappeared in revision 13992 after surviving more than 813 days. We found that the clones were modified over 83 times by 10 different developers and were finally removed when fixing bug id 172633. As another example, we found a clone genealogy that

Table 4. Correlation analysis results

	All	Columba	JDT	common	pig	HtmlUnit	jEdit	JFreeChart
Top 5	$a_{13}(0.553)$	$a_{27}(0.366)$	$a_{13}(0.632)$	$a_8(0.568)$	$a_{13}(0.494)$	$a_{13}(0.674)$	$a_{12}(0.385)$	$a_6(0.448)$
	$a_{12}(0.528)$	$a_{29}(0.353)$	$a_{12}(0.601)$	$a_4(0.563)$	$a_{12}(0.474)$	$a_8(0.647)$	$a_{29}(0.358)$	$a_{12}(0.446)$
	$a_8(0.481)$	$a_{28}(0.351)$	$a_4(0.562)$	$a_{10}(0.466)$	$a_4(0.302)$	$a_4(0.637)$	$a_{33}(0.358)$	$a_{11}(0.438)$
	$a_4(0.479)$	$a_{21}(0.307)$	$a_8(0.561)$	$a_7(0.456)$	$a_8(0.287)$	$a_{12}(0.628)$	$a_{21}(0.356)$	$a_5(0.415)$
	$a_{11}(0.458)$	$a_{25}(0.300)$	$a_{11}(0.493)$	$a_3(0.448)$	$a_{10}(0.247)$	$a_7(0.551)$	$a_{13}(0.347)$	$a_{10}(0.294)$
Bot. 5	$a_{31}(0.023)$	$a_{15}(0.031)$	$a_{23}(0.015)$	$a_{18}(0.048)$	$a_{24}(0.008)$	$a_{32}(0.051)$	$a_{15}(0.066)$	$a_1(0.041)$
	$a_{32}(0.018)$	$a_{18}(0.027)$	$a_{33}(0.013)$	$a_{17}(0.046)$	$a_{20}(0.006)$	$a_0(0.043)$	$a_9(0.047)$	$a_{17}(0.021)$
	$a_1(0.016)$	$a_0(0.027)$	$a_{24}(0.009)$	$a_{32}(0.026)$	$a_{25}(0.004)$	$a_{25}(0.040)$	$a_{16}(0.039)$	$a_{22}(0.016)$
	$a_{17}(0.014)$	$a_{10}(0.011)$	$a_{19}(0.009)$	$a_{24}(0.021)$	$a_{21}(0.004)$	$a_{21}(0.036)$	$a_0(0.036)$	$a_{18}(0.006)$
	$a_0(0.009)$	$a_{16}(0.007)$	$a_{20}(0.005)$	$a_{31}(0.012)$	$a_{26}(0.001)$	$a_1(0.014)$	$a_1(0.003)$	$a_{21}(0.006)$

contained 45 clones modified by 6 different developers in 20 different revisions, which survived 898 days before being removed.

6 Predicting the Survival Time of Clones

This section describes a decision-tree based model that predicts how long clones are likely to stay in a system. When building a training data set, we categorized a clone survival time into five categories: **very short-lived**, **short-lived**, **normal**, **long-lived**, and **very long-lived**. It is very important to find an unbiased binning scheme that converts the number of days a clone survived into a nominal label. If a binning scheme is chosen so that most vectors in the training data are put into a single bin, then the resulting prediction model is highly accurate by predicting always the same label. However, it is not useful because it cannot distinguish the survival time of clones.

To find an unbiased binning scheme, we explored two binning methods to convert a clone survival time into five categories. The first scheme is to gradually increase the size of a bin such that the bin size is larger than the preceding bin size: $bin_i = bin_{i-1} + 0.5 \times (i + 1) \times \chi$, where χ is the size of the first bin. For example, when χ is 50, the binning scheme is $\{ [0,50), [50, 125), [125, 225), [225, 350), [350, \infty) \}$. The second scheme is to uniformly assign a bin size to χ . For both schemes, we varied the size of the first bin χ to be 30, 40, 50, 60 and 70 and computed the entropy measure to assess distribution of the training data set across those bins. If the training feature vectors are equally distributed across five bins, the entropy should be 2.3219 ($= -\log_2 \frac{1}{5}$). If all vectors are localized in a single bin, the entropy will be 0. For each subject program, we then selected a binning scheme with the highest entropy score, which will distribute the training set as uniformly as possible across the five bins. Figure 5 shows the entropy score for different binning schemes. After selecting a binning scheme with the highest entropy score, the training data set is distributed across the selected bins as shown in Table 6.

We built prediction models using five different classifiers in the Weka toolkit (K Nearest Neighbor, J48, Naive Bayes, Bayes Network and Random Forest) on these data sets. The J48 decision tree-based classifier [29] combined with the

Table 5. Entropy measures for various binning schemes

project	entropy in incremental scheme					entropy in uniform scheme				
	χ value	30	40	50	60	70	30	40	50	60
Columba	1.797	1.954	1.796	1.860	1.955	1.256	1.474	1.769	1.958	1.945
Eclipse ¹	1.101	1.496	1.804	1.534	1.919	0.642	0.659	1.092	1.365	1.755
common	1.229	1.367	1.353	1.274	1.235	1.201	1.330	1.352	1.360	1.331
pig	1.979	2.201	2.061	2.040	1.826	1.512	2.106	2.139	2.172	2.067
HtmlUnit	2.009	1.943	1.662	2.157	2.036	1.724	1.945	2.009	2.065	2.102
jEdit	1.085	1.454	1.544	1.735	1.843	0.604	0.821	1.136	1.465	1.473
JFreeChart	1.654	1.535	1.882	1.535	1.339	0.846	0.728	0.769	1.535	1.535

¹ For Eclipse JDT, we tried additional χ values (80, 90, 100). For the incremental binning scheme, entropy = 2.227, 2.264, and 1.970 respectively, while using an uniform scheme, entropy = 1.823, 1.543, and 1.784.

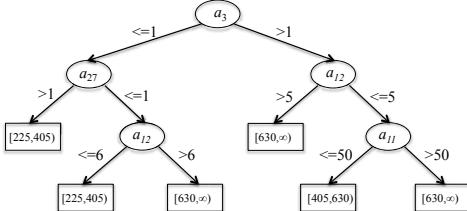
Table 6. Categorization of feature vectors based on a selected binning scheme

project	# of vectors	survival time (days)	# of genealogies for each category
Columba	102	1.1 ~ 1222.2	[0,60):18, [60,120):8, [120,180):9, [180,240):16, [240+):51
Eclipse	1826	68.7 ~ 2010.0	[0,90):204, [90,225):423, [225,405):340, [405,630):510, [630+):349
common	455	34.0 ~ 585.0	[0,40):324, [40,100):66, [100,180):16, [180,280):33, [280+):16
pig	422	30.0 ~ 536.9	[0,40):131, [40,100):91, [100,180):97, [180,280):31, [280+):72
HtmlUnit	425	6.9 ~ 2122.4	[0,60):125, [60,150):119, [150,270):63, [270,420):24, [420+):94
jEdit	245	13.3 ~ 2281.7	[0,70):22, [70,175):31, [175,315):31, [315,490):22, [490+):139
JFreeChart	219	11.1 ~ 415.0	[0,50):37, [50,125):2, [125,225):104, [225,350):38, [350+):38

¹ $[n, m):k$ means there are k number of vectors whose survival time lie in between n to m days.

bagging method [6] performed the best among these classifiers in terms of overall accuracy. J48 is an implementation of Quinlan's decision tree learner C4.5 [29] based on information entropy. At each node of the tree, it chooses an attribute that most effectively splits the data set into subsets. The attribute that results in the highest normalized information gain (difference in entropy) is used to split the data. Bagging is a bootstrapping method, proposed by L. Breiman [6] that generates multiple versions of a predictor and aggregates these predictors into a new predictor. Since our class label is nominal, the resulting predictor uses a voting scheme to produce a new class label. Figure 4 shows a J48 decision tree for Eclipse JDT core. This model considers factors such as the number of developers who modified clones and the dispersion of clones in a system to predict how long the clones are likely to survive in a system.

- a_3 : The number of *add* evolution patterns.
- a_{11} : The number of times that files containing clones were modified.
- a_{12} : The number of developers involved in maintaining clones.
- a_{27} : The number of unique methods that clones in the last version are located.

**Fig. 3.** An excerpt of a resulting decision tree for Eclipse JDT core

Project	Precision	Recall
Columba	58.1%	58.8%
Eclipse JDT core	79.4%	79.3%
Hadoop.common	74.5%	78.0%
Hadoop.pig	79.1%	79.1%
HtmlUnit	73.3%	73.6%
jEdit	62.0%	65.7%
JFreeChart	68.2%	70.3%
Total	75.7%	76.5%

Fig. 4. Prediction model

We use 10-fold cross validation to evaluate the prediction model based on two measures: (1) a weighted average precision, $\frac{\sum_{i=1}^n \frac{TP_i}{TP_i + FP_i} \times t_i}{\sum_{i=1}^n t_i}$ and (2) a weighted average recall, $\frac{\sum_{i=1}^n \frac{TP_i}{t_i}}{\sum_{i=1}^n t_i}$, when TP_i is the number of correct predictions of each class label i , FP_i is the number of incorrect predictions of i , and t_i is the total number of vectors with a class label i in the training data set. Table 6 summarizes

the weighted average precision and recall measures for each project. Our precision ranges from 58.1% to 79.4%, and our recall ranges from 58.8% to 79.3%. This result shows promise in using the attributes extracted from clone evolution data to predict a clone survival time.

7 Limitations

Our clone genealogy extractor (CGE) uses CCFinder to detect code clones and to map clones across versions [15]. CCFinder is a token-based clone detection technique that transforms tokens of a program according to a language-specific rule and performs a token-by-token comparison. CCFinder is recognized as a state of the art clone detector that handles industrial size programs; it is reported to produce higher recall although its precision is lower than some other tools. CCFinder does not detect *non-contiguous* clones and it is sensitive to reordering statements. This limitation leads to CGE's limitation in extracting clone genealogy data. If a programmer consistently modified an old clone group OG to create a new clone group NG , CCFinder does not find a cloning relationship between OG and NG if they do not share a contiguous token string greater than the size of $sim_{th} = (|OG.text| + |NG.text|)/2$. The absence of a cloning relationship can be mistakenly interpreted as a discontinuation of a lineage. Furthermore, CGE incorrectly counts the number of consistent change patterns in some cases, because CCFinder detects only a contiguous token string as a clone. For example, when code is inserted in the middle of one clone in a clone group, the existing clone group is broken into two new clone groups with shorter contiguous text, causing identification of two consistent patterns rather than one inconsistent change pattern. Similarly when the statements in a clone are reordered, such clones could be considered as removed because CCFinder may not be able to detect those clones.

We set the minimum token threshold of CCFinder to be 40 tokens and the similarity threshold sim_{th} for associating clones between consecutive versions as 0.8. Thus, we considered only the clones that are at least 40 tokens long and could map clones across versions only when the old and new clones are at least 80% similar according to CCFinder's equivalence criteria.

We extracted clone genealogies at a temporal granularity of major releases because CGE could not handle more than 1000 program versions as input. Studying clone evolution data at a finer temporal granularity such as check-in snapshots may provide more accurate evolutionary characteristics of long-lived clones. When studying the characteristics of clones, we did not consider the dispersion of clones in a class hierarchy or the refactorability of clones. Further investigation of such characteristics remains as future work.

8 Conclusions

Previous studies on code cloning indicate that clones are not necessarily harmful and that refactoring may not be always applicable to clones or be even beneficial

for them. As a first step toward selectively identifying clones to refactor, we conducted an empirical investigation into the characteristics of long-lived clones. Based on our prior work on clone genealogy extraction, we developed a method that takes a clone genealogy as input and generates a feature vector to encode its characteristics. By feeding the feature vectors to decision-tree based classification algorithms, we developed models that predict a clone survival time. The study found that the size of a clone, the number of clones in the same group, and the method-level distribution of clones are not strongly correlated with a clone survival time. However, the number of developers who modified clones and the time since the last addition or removal of a clone to its group are highly correlated with the survival time of clones. The survival time prediction model has 75.7% precision and 76.5% recall, showing promise in selectively identifying clones to remove.

References

1. Aversano, L., Cerulo, L., Penta, M.D.: How clones are maintained: An empirical study. In: CSMR 2007, pp. 81–90. IEEE Computer Society Press, Washington, DC, USA (2007)
2. Balazinska, M., Merlo, E., Dagenais, M., Lagüe, B., Kontogiannis, K.: Advanced clone-analysis to support object-oriented system refactoring. In: WCRE 2000, p. 98. IEEE Computer Society, Washington, DC, USA (2000)
3. Balint, M., Marinescu, R., Girba, T.: How developers copy. In: ICPC 2006, pp. 56–68. IEEE Computer Society, Washington, DC, USA (2006)
4. Baxter, I.D., Yahin, A., Moura, L., Sant’Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: ICSM 1998, p. 368. IEEE Computer Society, Washington, DC, USA (1998)
5. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.* 33(9), 577–591 (2007)
6. Breiman, L.: Bagging predictors. *Mach. Learn.* 24(2), 123–140 (1996)
7. Cordy, J.R.: Comprehending reality practical barriers to industrial adoption of software maintenance automation. In: IWPC 2003, p. 196. IEEE Computer Society, Washington, DC, USA (2003)
8. Cottrell, R., Chang, J.J.C., Walker, R.J., Denzinger, J.: Determining detailed structural correspondence for generalization tasks. In: ESEC-FSE 2007, Dubrovnik, Croatia, pp. 165–174. ACM, New York (2007)
9. Ducasse, S., Rieger, M., Demeyer, S.: A language independent approach for detecting duplicated code. In: ICSM 1999, p. 109. IEEE Computer Society, Washington, DC, USA (1999)
10. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Reading (2000)
11. Garcia, J., Popescu, D., Edwards, G., Medvidovic, N.: Identifying architectural bad smells. In: CSMR 2009, pp. 255–258. IEEE Computer Society, Washington, DC, USA (2009)
12. Godfrey, M.W., Svetinovic, D., Tu, Q.: Evolution, growth, and cloning in linux, a case study. In: CASCON 2000 (2000)

13. Higo, Y., Kamiya, T., Kusumoto, S., Inoue, K.: Refactoring support based on code clone analysis. In: Bomarius, F., Iida, H. (eds.) PROFES 2004. LNCS, vol. 3009, pp. 220–233. Springer, Heidelberg (2004)
14. Howard Johnson, J.: Identifying redundancy in source code using fingerprints. In: CASCON 1993, pp. 171–183. IBM Press (1993)
15. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A multilingual token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.* 28(7), 654–670 (2002)
16. Kapser, C., Godfrey, M.W.: Aiding comprehension of cloning through categorization. In: IWPSE 2004, Washington, DC, USA, pp. 85–94. IEEE Computer Society Press, Los Alamitos (2004)
17. Kapser, C., Godfrey, M.W.: Improved tool support for the investigation of duplication in software. In: ICSM 2005: Proceedings of the 21st IEEE International Conference on Software Maintenance, Washington, DC, USA, pp. 305–314. IEEE Computer Society Press, Los Alamitos (2005)
18. Kapser, C., Godfrey, M.W.: "Cloning Considered Harmful" Considered Harmful. In: WCRE 2006, pp. 19–28. IEEE Computer Society, Washington, DC, USA (2006)
19. Kim, M., Bergman, L., Lau, T., Notkin, D.: An ethnographic study of copy and paste programming practices in oopl. In: ISESE 2004, pp. 83–92. IEEE Computer Society, Washington, DC, USA (2004)
20. Kim, M., Sazawal, V., Notkin, D., Murphy, G.: An empirical study of code clone genealogies. In: ESEC/FSE-13, pp. 187–196. ACM Press, New York (2005)
21. Komondoor, R., Horwitz, S.: Semantics-preserving procedure extraction. In: POPL 2000, pp. 155–169. ACM Press, New York (2000)
22. Komondoor, R., Horwitz, S.: Effective, automatic procedure extraction. In: IWPC 2003, p. 33. IEEE Computer Society, Washington, DC, USA (2003)
23. Koni-N'-Sapu, G.G.: A scenario based approach for refactoring duplicated code in object-oriented systems. Master's thesis, University of Bern (2001)
24. Krinke, J.: A study of consistent and inconsistent changes to code clones. In: WCRE 2007, pp. 170–178. IEEE Computer Society, Washington, DC, USA (2007)
25. Lague, B., Proulx, D., Mayrand, J., Merlo, E.M., Hudepohl, J.: Assessing the benefits of incorporating function clone detection in a development process. In: ICSM 1997, p. 314. IEEE Computer Society, Washington, DC, USA (1997)
26. LaToza, T.D., Venolia, G., DeLine, R.: Maintaining mental models: a study of developer work habits. In: ICSE 2006, pp. 492–501. ACM, New York (2006)
27. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In: OSDI 2004, pp. 289–302 (2004)
28. Moha, N., Guéhéneuc, Y.-G., Le Meur, A.-F., Duchien, L.: A domain analysis to specify design defects and generate detection algorithms. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008/ETAPS 2008. LNCS, vol. 4961, pp. 276–291. Springer, Heidelberg (2008)
29. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann, San Francisco (1993)
30. Rajapakse, D.C., Jarzabek, S.: Using server pages to unify clones in web applications: A trade-off analysis. In: ICSE 2007, pp. 116–126. IEEE Computer Society, Washington, DC, USA (2007)
31. Reza, F.M.: An Introduction to Information Theory. Dover Publications, Inc., New York (1996)

32. Rodgers, J.L., Nicewander, W.A.: Thirteen ways to look at the correlation coefficient. *The American Statistician* 42, 59–66 (1988)
33. Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A.: Jdeodorant: Identification and removal of type-checking bad smells. In: CSMR 2008, pp. 329–331. IEEE Computer Society Press, Washington, DC, USA (2008)
34. Tsantalis, N., Chatzigeorgiou, A.: Identification of extract method refactoring opportunities. In: CSMR 2009, pp. 119–128. IEEE Computer Society, Washington, DC, USA (2009)
35. Tsantalis, N., Chatzigeorgiou, A.: Identification of move method refactoring opportunities. *IEEE Trans. Softw. Eng.* 35(3), 347–367 (2009)

Where the Truth Lies: AOP and Its Impact on Software Modularity

Adam Przybyłek

University of Gdańsk, Department of Business Informatics,
Piaskowa 9, 81-824 Sopot, Poland
adam@univ.gda.pl

Abstract. Modularity is the single attribute of software that allows a program to be intellectually manageable [29]. The recipe for modularizing is to define a narrow interface, hide an implementation detail, keep low coupling and high cohesion. Over a decade ago, aspect-oriented programming (AOP) was proposed in the literature to “modularize the un-modularizable” [24]. Since then, aspect-oriented languages have been providing new abstraction and composition mechanisms to deal with concerns that could not be modularized because of the limited abstractions of the underlying programming language. This paper is a continuation of our earlier work [32] and further investigates AO software with regard to coupling and cohesion. We compare two versions (Java and AspectJ) of ten applications to review AOP within the context of software modularity. It turns out that the claim that “the software built in AOP is more modular than the software built in OOP” is a myth.

Keywords: AOP, modularization, separation of concerns.

1 Introduction

Modularity is a key concept that programmers wield in their struggle against the complexity of software systems. Modularization is the process of decomposing a system into logically cohesive and loosely-coupled modules that hide their implementation from each other and present services to the outside world through a well-defined interface [3, 28, 30, 43]. Cohesion is the “intramodular functional relatedness” and describes how tightly bound the internal elements of a module are to one another, whereas coupling is “the degree of interdependence between modules” [43]. Modularization makes it possible to reason about every module in isolation, such that when a small change in requirements occurs, it will be possible to go to one place in code to make the necessary modifications [11].

In practice, modularization corresponds with finding the right decomposition of a problem [13]. However, in traditional programming languages, no matter how well a software system is decomposed into modules, there will always be concerns (typically non-functional ones) whose code cuts across the chosen decomposition [27]. The implementations of these crosscutting concerns will necessarily be spread over different modules, which has a negative impact on maintainability and reusability.

Such concerns are called crosscutting concerns. The presented problem is known as the “tyranny of the dominant decomposition” [41] and several techniques have been invented to overcome it. The most prominent among them is AOP [23].

2 Motivations and Goals

Whenever a new technology is proposed, it has to prove its superiority over existing competitors. AOP emerged as a new paradigm to modularize the concerns whose implementations would otherwise have been spread throughout the whole application, because of the limited abstractions of the underlying programming languages. Since then, several studies [15, 16, 17, 19, 35, 36] have suggested that AOP is successful in modularizing crosscutting concerns. Unfortunately, these studies either wrongly identify modularization with the lexical SoC offered by AOP, or wrongly measure coupling in AO systems. Indeed, in our previous study [32] we adapted the CBO metric to AOP and investigated implementations of the 23 GoF design patterns. We found no cases in which an AO implementation was more modular than its OO counterpart. Since these results cannot be generalized beyond simple academic examples, we continue our earlier work and evaluate real-life systems.

3 Modularity Metrics

3.1 Measurement System

In order to compare software modularity between the OO and AO paradigm, we used the G-Q-M (Goal-Question-Metric) approach [2]. G-Q-M defines a measurement system on three levels (Fig. 1) starting with a goal. The goal is refined in questions that break down the issue into quantifiable components. Each question is associated with metrics that, when measured, will provide information to answer the question. Our goal is to compare AO and OO systems with respect to software modularity from the viewpoint of the developer.

In our previous paper [32], we argued for measuring modularity with the help of coupling and cohesion. This pair of attributes was firstly suggested to measure software modularity by Yourdon & Constantine [43] as part of their structured design methodology and then it was adapted to OO methodology by Coad & Yourdon [12], Booch [3], and Meyer [28]. Also, several empirical studies [6, 7, 20, 31] confirm that improvements in coupling and cohesion are linked to improved modularity.

Despite coupling and cohesion having been concepts in software design for almost 50 years, we still do not have widely-accepted metrics for them. In our previous paper [32], we supported CBO (Coupling Between Object classes) and LCOM (Lack of Cohesion in Methods), adapted from the Chidamber & Kemerer (CK) metrics suite [10]. CBO is a count of the number of other modules to which a module is coupled. Two modules are coupled when methods declared in one module use methods or instance variables of the other module [10]. LCOM is the degree to which methods within a module are related to one another. It is measured as the number of pairs of methods working on different attributes minus pairs of methods working on at least one shared attribute (zero if negative).

CBO and LCOM complement each other, and because of their dual nature, they are useful only when analyzed together. Attempting to optimize a design with respect to CBO alone would trivially yield to a single giant module with no coupling. However, such an extreme solution can be avoided by considering also the antagonistic attribute LCOM (which would yield inadmissibly high values in the single-module case) [20].

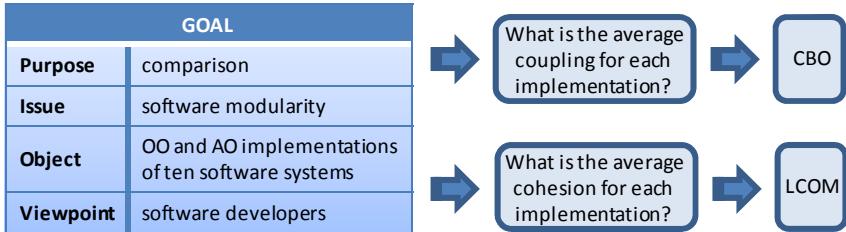


Fig. 1. Goal-Question-Metric

Since AOP provides new programming abstractions, existing OO measures cannot be directly applied to AO software. The efforts to make the CK metrics suite applicable to AO software were originated by Ceccato & Tonella [9] and Sant'Anna et al. [35]. Zhao [45] complemented their work by specifying the coupling dependencies in a formal way. Their general suggestion is to treat advices as methods and to consider introductions as members of the aspect that defines them. Although this suggestion is enough to adapt LCOM, the adjustment of CBO requires further explanation.

Coupling is a more complex attribute in AO systems, because new programming constructs introduce novel kinds of coupling relationships. We found that the existing coupling metrics [9, 35, 45] take into account only syntactic dependency. Syntactic dependency occurs when there is a direct reference between modules, such as inheritance or composition. However, in AO programs there is another kind of dependency that is not so easy to realize because it occurs without explicit references in the code. Ribeiro et al. [34] called this kind of coupling as semantic dependency. In our earlier work [32], we proposed a metric that considers this subtle kind of coupling. Our CBO metric considers a module M to be coupled to N if (in parentheses, we provide the abbreviations for the dependencies):

- M accesses attributes of N (A);
- M calls methods of N (M);
- M potentially captures messages to N (C);
- Messages to M are potentially captured by N (C_by);
- M declares an inter-type declaration for N (I);
- M is affected by an inter-type declaration declared in N (I_by);
- M uses pointcuts of N, excluding the case where N is an ancestor of M (P).

The C_by and I_by dependencies are semantic.

3.2 Rationale for Semantic Dependency

To construct our metric, we extrapolated the original CBO definition according to the question that underlies coupling: “How much of one module must be known in order to understand another module?” [43]. The syntactic dependencies (i.e. A, M, C, I, P) occurring in our metric do not raise any doubts even among proponents of AOP. Nevertheless, the debate on our previous paper [32], during and after the panel discussion at ENASE10, demonstrated that further explanation of the semantic dependencies is required. It suffices to show that for understanding a given module M, we have to analyze N if the C_by or I_by dependency exists between M and N.

Suppose there are two modules as shown in Listing 1. Next, we send the inc(5) message to an instance of M. If we analyze M without considering the C_by dependency from M to N1, we will deduce (following program control flow) that the result is 6. However, the result is actually 11, and analyzing N1 is necessary to compute it correctly.

```
public class M {
    public int inc(int x) {
        return ++x;
    }
}
public aspect N1 {
    int around(int i):
        execution( int M.inc(int) ) && args(i) {
            return proceed(2*i);
    }
}
```

Listing 1. The C_by dependency

Now, suppose two new modules were added as shown in Listing 2. This time the same message is being sent to an instance of SubM. Once again, if we analyze M without considering the I_by dependency from SubM to N2, we will deduce an incorrect result. The correct is 20.

```
public aspect N2 {
    public int subM.inc(int x) {
        return x+10;
    }
}
public class SubM extends M {}
```

Listing 2. The I_by dependency

4 Empirical Study

4.1 Assessment Procedures

We compared two versions (Java and AspectJ) of 10 different systems: Telestrada, Pet Store, CVS Core, ElImp, Health Watcher, JHotDraw, HyperCast, Prevayler,

Berkeley DB, and HyperSQL Database. To the best of our knowledge, these are all systems that have been implemented in both Java and AspectJ. They have also been widely used by other researchers to evaluate their work in the area of AOP.

The assessment of both versions bases on the application of metrics that quantify two fundamental modularity attributes, namely coupling and cohesion. In addition, we supplement our study by size metrics. Table 1 overviews the employed metrics and associates them with the attributes measured by each one of them. Detailed description of the coupling metric is provided in Sect. 3.1.

The gathering of data for the metrics was automated through the use of the extended version of AOPmetrics [37]. We extended AOPmetrics to support the CBO metric as defined in the previous section, except for capturing C and C_by (available at: <http://przybylek.wzr.pl/AOP/>). This is due to some inherent bugs in AOPmetrics [32]. Hence, we manually revised the CBO measures.

Table 1. Metric definitions

Attributes	Metrics	Definitions
Size	Vocabulary Size	Number of modules (classes, interfaces, and aspects) of the system
	Lines of Code	Number of lines in the text of the system's source code
Modularity	Coupling Between Object classes	Number of other modules to which a module is coupled
	Lack of Cohesion in Methods	Number of pairs of methods/advices working on different attributes minus pairs of methods working on at least one shared attribute (zero if negative)

4.2 Selected Systems

Our study uses ten real-world systems from different domains and of varying sizes. These are:

- Telestrada, which is a traveler information system being developed for a Brazilian national highway administrator. It allows its users to register and visualize information about Brazilian roads;
- Pet Store, which is a demo for the J2EE platform that is representative of existing e-commerce applications;
- CVS Core, which is an Eclipse Plugin that implements the basic functionalities of a CVS client, such as checkin and checkout of a system stored in a remote repository;
- EIImp, which is an Eclipse Plugin that supports collaborative software development for distributed teams;
- Health Watcher, which is a web-based information system that was developed by Soares [36] for the healthcare bureau of the city of Recife, Brazil. The system aims to improve the quality of services provided by the healthcare institution, allowing

citizens to register complaints regarding health issues, and the healthcare institution to investigate and take the required actions. It involves a number of recurring concerns and technologies common in day-to-day software development, such as GUI, concurrency, RMI, Servlets and JDBC;

- JHotDraw (www.jhotdraw.org), which is a framework for technical and structured 2D graphics. Its design relies heavily on some well-known design patterns. JHotDraw's original authors were Gamma & Eggenschwiler;
- HyperCast (www.hypercast.org), which is software for developing protocols and application programs for application-layer overlay networks. It supports a variety of overlay protocols, delivery semantics and security schemes, and has a monitor and control capability. It was developed at the University of Virginia in cooperation with the Microsoft Corporation;
- Prevayler (www.prevayler.org), which is an object persistence library for Java. It is an implementation of the Prevalent System design pattern, in which business objects are kept live in memory and transactions are journaled for system recovery. Business object must be serializable, i.e., implement the `java.io.Serializable` interface, and deterministic, i.e., given an input, the object's methods must always return the same output;
- Berkeley DB Java Edition (oracle.com/technology/products/berkeley-db), which is a database system that can be embedded in other applications as a fast transactional storage engine. It stores arbitrary key/data pairs as byte arrays and supports multiple data items for a single key. Berkeley DB provides the underlying storage and retrieval system of several LDAP servers, database systems and many other applications;
- HyperSQL Database (hsqldb.org), which implements a relational database management system. It offers a small and fast database engine which supports both in-memory and disk-based tables. HSQLDB is currently being used as a database and persistence engine in many projects, such as Mathematica and OpenOffice.

All of these systems were originally implemented in Java and, afterwards, were refactored using AspectJ, so that the code responsible for some crosscutting concerns was moved to aspects. In each case, code refactoring was done by proponents of AOP to present the benefits of AOP over OOP.

In the first four systems, aspects were used to implement exception handling [8], [16]. Exception-handling is known to be a global design issue that affects almost all system modules, mostly in an application-specific manner.

The next work (Health Watcher) [19], [36] goes beyond refactoring of exception handling including concerns such as data persistence, concurrency and distribution (basic remote access to system services using Java RMI). Both the OO and AO designs of the Health Watcher system were developed with modularity and changeability principles as main driving design criteria.

AJHotDraw (ajhotdraw.sourceforge.net) is an aspect-oriented refactoring of JHotDraw with regard to persistence, design policies contract enforcement and undo command. It was started to experiment with the feasibility of adopting aspect-oriented solutions in existing software and demonstrate the strategies proposed by research of the Software Evolution Research Lab of Delft University of Technology in the

Netherlands. The aims, objectives and experience of the AJHotDraw project are summarized by Marin et al. [26].

Sullivan et al. [40] encountered two types of development problems when refactoring logging and event notification in HyperCast. First, the tight coupling between aspects and method names prevented the development of aspects in parallel with primary code refactoring, because the aspects could only be developed after inspecting the core concerns. Second, they found cases where joinpoints were not accessible, because AspectJ supports specifying joinpoints at the method call level and data member level, but not at the if or switch statement level. Next, they reimplemented the base version using AspectJ and crosscutting interfaces (XPI). What distinguishes that particular release, is the lack of introductions used. In our experiment, we evaluate the improved version.

Prevayler was refactored using AspectJ and horizontal decomposition by Godil & Jacobsen [18]. The horizontal decomposition principles were proposed by Zhang & Jacobsen [44] to guide the AO refactoring and implementation of complex software systems. The refactored code includes persistence, transaction, query, and replication management [18].

Störzer et al. [38] refactored version 1.8.0 of HSQLDB (sourceforge.net/projects/ajhsqldb/). They started with an accepted catalog of well-known crosscutting concerns and then tried to find classes, methods or fields related to the respective concerns. They used manual semantics-guided code inspection supported by Feature Exploration and analysis tool to find a relevant crosscutting code. They discovered and refactored many standard crosscutting concerns, including Logging, Tracing, Exception Handling, Caching, Pooling and Authentication/Authorization. When becoming familiar with the source code, they also found some application specific aspects, for example trigger firing or checking constraints before certain operations are performed [39].

By analyzing the domain, manual, configuration parameters, and source code, Kästner et al. [22] identified many parts of Berkeley DB that represented increments in program functionality that were candidates to be refactored into features. These features are implicit in the original code. They vary from small caches to entire transaction or persistence subsystems. All identified features represent program functionality, as a user would select or deselect them when customizing a database system. From these features, they chose 38 and manually refactored one feature after another (wwwiti.cs.uni-magdeburg.de/iti_db/berkeley/). They used various OOP-to-AOP refactoring techniques, including Extract Introduction, Extract Beginning and Extract End, Extract Before/After Call, Extract Method, and Extract Pointcut [22].

4.3 Experimental Results

Table 2 shows the obtained results for both size metrics, vocabulary size (n) and LOC, and both modularity metrics, CBO and LCOM. For all the employed metrics, a lower value implies a better result. The fourth and fifth column presents the mean values of the measures, over all modules per system. Rows labeled ' Δ ' indicate the percentage difference between the OO and AO implementations relative to each metric. A positive value means that the original version performs better, whereas a negative value indicates that the refactored version exhibits better results.

Table 2. Results for size, coupling and cohesion metrics

		n	LOC	CBO	LCOM
Telestrada	OO	233	3424	0,81	1,86
	AO	242(18)	3350	0,95	2,17
	Δ	4%	-2%	18%	16%
PetStore	OO	345	17798	2,32	20,63
	AO	382(37)	17914	2,76	20,19
	Δ	11%	1%	19%	-2%
CVS	OO	257	18876	5,76	71,31
	AO	261(4)	19423	higher	73,90
	Δ	2%	3%	x	4%
Elmp	OO	123	8708	1,84	1,53
	AO	126(3)	9041	higher	1,68
	Δ	2%	4%	x	10%
Health Watcher	OO	88	6096	3,19	9,24
	AO	103(12)	5768	4,20	7,63
	Δ	17%	-5%	32%	-17%
JHotDraw	OO	398	22724	3,57	75,04
	AO	438(31)	23167	3,66	65,70
	Δ	10%	2%	3%	-12%
Hypercast	OO	370	50492	3,31	67,24
	AO	391(7)	51207	3,42	67,00
	Δ	6%	1%	4%	-0,4%
Prevayler	OO	167	5043	1,87	9,31
	AO	168(55)	4179	2,56	7,01
	Δ	1%	-17%	37%	-25%
Berkeley DB	OO	340	41651	4,38	126,31
	AO	452(107)	38770	4,73	78,21
	Δ	33%	-7%	8%	-38%
HSQLDB	OO	402	80736	4,11	226,91
	AO	413(25)	76210	4,12	247,30
	Δ	3%	-6%	0,3%	9%

In the case of both Eclipse Plugins, their refactored code is not publicly available, so we based our analysis on the measurements carried out by Castor et al. [8]. However, since they do not consider all types of coupling, we cannot present the exact CBO values. We can only say that coupling is greater for the refactored systems.

Contradicting the general intuition that AOP makes programs smaller, the refactored versions are larger with regard to the LOC metric in half the cases. Nevertheless, the increase ranges between 1% and 4%. In the remaining half, differences are greater than 5% (except for Telestrada) in favor of AOP.

The average coupling between modules is significantly higher in most of the refactored versions. For the refactored versions of Prevayler and Health Watcher, it is more than 30% higher than for the corresponding OO releases. Only for HSQLDB, JHotDraw and HyperCast is the increase rather slight. The higher coupling is the result of introducing new constructs intrinsic for AOP. In a typical scenario during AO refactoring, the coupling generated by explicit method call is replaced by the coupling generated by implicit advice triggering. Since an implicit advice triggering is associated with C and C_by in our CBO metric, the overall coupling grows. In addition, Filho et al. found [16] that new coupling was introduced when exception-handler aspects had to capture contextual information from classes.

Although the obtained results were as expected due to the above presented theoretical considerations, they contradict the outcomes achieved in several earlier studies. The advocates of AOP claim that the refactored versions of Telestrada [8, 16], Pet Store [8, 16], CVS Core Plugin [8, 16], EIImp Plugin [8], Health Watcher [19, 36], and Prevayler [18] exhibit lower coupling. However, they take into account only a subset of the dependencies that generate coupling in AO systems. Hence, the coupling measured with their metrics is underestimated.

The Lack of Cohesion in Methods is the metric for which the impact of AOP has remained unclear. For the refactored versions of Berkeley DB, Prevayler, Health Watcher and JHotDraw, the average LCOM is respectively 38%, 25%, 17%, and 12% lower than for the corresponding original versions. On the other hand, the average LCOM grew by 16% in the refactored version of Telestrada, 10% in the EIImp Plugin, and 9% in HSQLDB. A partial explanation for this increase is the large number of methods that were created to expose join points (e.g. try-catch blocks in loops, etc.) that AspectJ can capture [21]. As discussed in [8], these new methods are not part of the implementation of the exception-handling concern but a direct consequence of using aspects to implement this concern. The average LCOM varied (positively or negatively) by less than 4% in the refactored versions of the remaining systems.

It is worth mentioning that most researchers compare aggregate coupling and cohesion between an OO and AO version of the same system. Aggregate coupling (cohesion) for a system is calculated as the sum of coupling (cohesion) taken over all modules. Hence, it can be derived from Table 2 as multiplication of the average value by vocabulary size. It should be also noted that the original versions perform better with regard to the aggregate coupling and cohesion, since the measures of vocabulary size grew in all cases, due to the introduction of aspects. Nevertheless, aggregate coupling does not satisfy the second axiom of Fenton & Melton [14] for coupling measures. That axiom states that system coupling should be independent from the number of modules in the system. If a module is added and shows the same level of pairwise coupling as the already existing modules, then the coupling of the system remains constant.

The obtained results confirm our previous findings. In [32], we compared Java and AspectJ implementations of 23 GoF design patterns. There was no pattern whose AO implementations would exhibit lower coupling, while 22 patterns presented lower coupling in the original implementations. With regard to cohesion, the OO implementations were superior in 9 cases, while the AO ones in 6 cases. 8 patterns exhibited the same cohesion in both implementations.

5 Threats to Validity

There are a number of limitations of this study that are worth stating. Firstly, we narrow software modularity to cohesion and coupling, despite of many other factors assigned to it. Nevertheless, cohesion and coupling are the concepts that lie at the heart of software modularity and are considered as main factors related to the goodness of modularization [3, 6, 7, 20, 28, 31]. The causal effect of AOP on software modularity was demonstrated in our earlier study [33].

Secondly, we could be criticised for applying metrics that are theoretically flawed. Briand et al. demonstrate [4] that LCOM is neither normalized nor monotonic. Normalization is intended to allow for comparison between modules of different size. To avoid this anomaly we weighted LCOM by the number of methods. Monotonicity states that adding a method which shares an attribute with any other method of the same module, must not increase LCOM. If we drop the very rare case where the methods of a module do not reference any of the attributes, the monotonicity anomaly disappears. The other problem with LCOM is that it does not differentiate modules well [1]. This is partly due to the fact that LCOM is set to zero whenever there are more pairs of methods which use an attribute in common than pairs of methods which do not [4]. In addition, the presence of access methods artificially decreases this metric. Access methods typically reference only one attribute, namely the one they provide access to, therefore they increase the number of pairs of methods in the class that do not use attributes in common [4]. The CBO metric also indicates inherent weakness. Briand et al. illustrate [5] that merging two unconnected modules may affect the overall coupling. Nevertheless, CBO as well as LCOM are widely applied and have been validated in many empirical studies [1, 5], and [6].

Thirdly, the applied metrics address only one possible dimension of cohesion and coupling. Moreover, CBO implicitly assumes that all basic couples are of equal strength [20]. In addition, it takes a binary approach to coupling between modules: two modules are either coupled or not. Multiple connections to the same module are counted as one [5]. In our defence we would point out that the OO community has yet to arrive at a consensus about the appropriate measurement of coupling and cohesion. The interested reader is referred to [4, 5], and [20] where an extensive surveys have been presented.

Finally, we could be criticised for generalizing findings from AspectJ to AOP. In our defence, most of the claims about the superiority of the AO modularization have been made in the context of AspectJ. It also should be noted that AspectJ is the only production-ready general purpose AO language.

To conclude, we are well aware that CBO and LCOM suffer from several disadvantages. We also known that the modularity evaluated in our setting may differ from the real modularity. The reason is that, it is not yet clear neither how to best measure attributes such as coupling and cohesion, nor how to compare modularity between systems that were developed in different paradigms. Nevertheless, the cases investigated provide enough evidence to challenge the claim that AOP improves software modularity.

6 Related Work

There are few studies focusing on the quantitative evaluation of the AO modularization. Sant'Anna et al. [35] conducted a semi-controlled experiment to compare the use of an OO approach (based on design patterns) and an AO approach to implement Portalware (about 60 modules and over 1 KLOC), a multi-agent system. Portalware is a web-based environment that supports the development and management of Internet portals. The collected metrics show that the AO version incorporates modules with higher coupling and lower cohesion. Their coupling metric is broader than the original CBO in the sense that it additionally counts modules declared in formal parameters, return types, throws declarations and local variables. However, it is not complete, since it does not take into account either the semantic dependencies, or the dependency that occurs when an advice refers to a pointcut defined in other, non-ancestor module.

The same suite of metrics was used by Garcia et al. [17] to compare the AO and OO implementations of the Gang-of-Four design patterns. They performed two studies, one on the original implementations from Hannemann & Kiczales and the other on the implementations with introduced changes. These changes were introduced because the H&K implementations encompassed few participant classes to play pattern roles [17]. Garcia and his team concluded that “the use of aspects helped to improve the coupling and cohesion of some pattern implementations.” However, such conclusion may be misleading, according to the metrics they collected. The measures before the application of the changes exhibit that only Composite and Mediator present lower coupling for the AO solutions. The implementations of Adapter and State have the same coupling in both paradigms. In the case of the other patterns, the OO solutions indicate lower coupling. The superiority of OO solutions decreased a little after the changes were introduced. Although the AO implementations of Observer, Chain of responsibility, State and Visitor became better with respect to coupling than their OO counterparts, there are still 16 patterns for which the OO implementations provide superior results. With regard to cohesion, the OO implementations were also superior in most cases. They analyzed the absolute (aggregate) values.

Other studies can be classified into 2 groups. In the first group [8, 16, 19, 25], new kinds of coupling introduced by pointcuts are not considered at all. In the second group [21, 42], the coupling introduced by a pointcut is considered only if a module is explicitly named by the pointcut expression.

Greenwood et al. [19] chose the Health Watcher system as the base for their study. Their evaluation focused upon ten releases of the system, which underwent a number of typical maintenance tasks, including: refactorings, functionality increments, extensions of abstract modules and more complex system evolutions. Some of the crosscutting concerns were “aspectized” from the first release, while others were modularized as new HW versions were released. They found that modularity was improved with AOP. The average “coupling” as well as cohesion were enhanced by 17% in the initial version, and by 23% and 21% in the 10th release.

Madeyski & Szała [25] examined the impact of AOP on software development efficiency and design quality in the context of a web-based manuscript submission and a review system (about 80 modules and 4 KLOC). Three students took part in their study. Two of them developed the system (labeled as OO1 and OO2) using Java, whilst one implemented the system using AspectJ. The observed results show that the AO version is 24% better than the others with regard to average “coupling” and it is 60% (3%) better than OO1 (OO2) with regard to average cohesion.

Filho et al. [8, 16] refactored to AOP four systems: Telestrada, Pet Store, CVS, and EIImp. The average “coupling” was decreased by 6%, 9%, and 1% for the first three systems and increased by 2% for the last system. Nevertheless, Filho et al. [18] were aware that their study missed some coupling dependencies introduced by AOP: “a closer examination on the code (...) reveals a subtle kind of coupling that is not captured by the employed metrics.”

The Telestrada and Pet Store systems were also used by Hoffman & Eugster. In their study [21], they calculated two coupling metrics, namely CBM and CIM. However, since CBM and CIM are not simply additive, the results are difficult to interpret.

Tsang et al. [42] compared AO vs. OO solutions in the context of real time traffic simulator. They found that aspects improved modularity by reducing “coupling” and cohesion. They considered aspects coupled to classes only if the aspects explicitly named the classes. “For instance, if we have the joinpoint call(* *(..)), then the aspect is not coupled to any classes. However, if we have the joinpoint call(void Test.methodName(..)), then the aspect is coupled to Test.” In the conclusion of their work, they recommend the use of wildcards to maximize modularity improvements. Following this reasoning, one could recommend to replace the previous pointcut by call(void Test.methodNam*(..)), where ‘*’ instead of ‘e’ eliminates “coupling”.

7 Summary

This paper presents an empirical study in which we compare OO and AO implementations of ten software systems with respect to modularity. The evaluation is performed using the CBO and LCOM metrics from the CK suite, which were adapted to AOP. We hope that this paper regenerates some discussion about the role of AOP in software development.

The contribution of our work are twofold. Firstly, we gave a rationale for our coupling metric. At the same time, we argued why the existing coupling metrics are invalid for evaluating AO systems – they do not take into account all the composition mechanisms offered by the underlying paradigm. Secondly, we found that there is no evidence that AOP promotes better modularity of software than OOP. The OO implementation of every system exhibits lower average coupling. With regard to average cohesion the OO implementations are superior in 4 cases, while the AO ones in 6 cases. As far as we know, this is the first presentation of empirical evidence to this effect on real-life systems.

References

1. Basili, V.R., Briand, L.C., Melo, W.L.: A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering* 22(10), 751–761 (1996)
2. Basili, V.R., Caldiera, G., Rombach, H.D.: Goal Question Metric Approach. In: *Encyclopedia of Software Engineering*, pp. 528–532. John Wiley & Sons, Inc., Chichester (1994)
3. Booch, G.: *Object-oriented Analysis and Design with Applications*. Benjamin-Cummings, Redwood City (1994)
4. Briand, L.C., Daly, J.W., Wüst, J.: A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Softw. Engg.* 3(1), 65–117 (1998)
5. Briand, L.C., Daly, J.W., Wüst, J.K.: A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering* 25(1), 91–121 (1999)
6. Briand, L.C., Morasca, S., Basili, V.R.: Defining and Validating Measures for Object-Based High-Level Design. *IEEE Trans. Softw. Eng.* 25(5), 722–743 (1999)
7. Briand, L.C., Wüst, J.K., Lounis, H.: Replicated Case Studies for Investigating Quality Factors in Object-Oriented Designs. *Empirical Software Eng* 6(1), 11–58 (2001)
8. Castor, F., Cacho, N., Figueiredo, E., Garcia, A., Rubira, C.M., de Amorim, J.S., da Silva, H.O.: On the modularization and reuse of exception handling with aspects. *Softw. Pract. Exper.* 39(17), 1377–1417 (2009)
9. Ceccato, M., Tonella, P.: Measuring the Effects of Software Aspectization. In: 1st Workshop on Aspect Reverse Engineering, Delft, Netherlands (2004)
10. Chidamber, S.R., Kemerer, C.F.: A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.* 20(6), 476–493 (1994)
11. Cline, M., Lomow, G., Girou, M.: *C++ FAQs*. Addison-Wesley, Reading (1998)
12. Coad, P., Yourdon, E.: *Object-Oriented Analysis*. Prentice-Hall, Englewood Cliffs (1991)
13. De Win, B., Piessens, F., Joosen, W., Verhanneman, T.: On the importance of the separation-of-concerns principle in secure software engineering. In: ACSA Workshop on the Application of Engineering Principles to System Security Design, Boston, Massachusetts (2002)
14. Fenton, N., Melton, A.: Deriving Structurally Based Software Measures. *J. Syst. Software* 12, 177–187 (1990)
15. Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F., Dantas, F.: Evolving software product lines with aspects: An empirical study on design stability. In: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany (2008)
16. Filho, F.C., Cacho, N., Figueiredo, E., Maranhão, R., Garcia, A., Rubira, C.M.: Exceptions and aspects: the devil is in the details. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Portland, Oregon (2006)
17. Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., von Staa, A.: Modularizing design patterns with aspects: a quantitative study. In: Proceedings of the 4th international Conference on Aspect-Oriented Software Development (AOSD 2005), Chicago, Illinois (2005)
18. Godil, I., Jacobsen, H.: Horizontal decomposition of Prevayler. In: The 2005 Conference of the Centre For Advanced Studies on Collaborative Research, Toronto, Canada (2005)

19. Greenwood, P., Bartolomei, T., Figueiredo, E., Dósea, M., Garcia, A.F., Cacho, N., Sant'Anna, C., Soares, S., Borba, P., Kulesza, U., Rashid, A.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 176–200. Springer, Heidelberg (2007)
20. Hitz, M., Montazeri, B.: Measuring Coupling and Cohesion in Object-Oriented Systems. In: Proceedings of the 3rd International Symposium on Applied Corporate Computing, Monterrey, Mexico (1995)
21. Hoffman, K., Eugster, P.: Bridging Java and AspectJ through explicit join points. In: 5th international Symposium on Principles and Practice of Programming in Java (PPPJ 2007), Lisboa, Portugal (2007)
22. Kästner, C., Apel, S., Batory, D.: A Case Study Implementing Features using AspectJ. In: 11th International Conference of Software Product Line Conference (SPLC 2007), Kyoto, Japan (2007)
23. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Cristina Lopes, C., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. In: Liu, Y., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
24. Lesiecki, N.: Improve modularity with aspect-oriented programming (2002), <http://www.ibm.com/developerworks/library/j-aspectj/>
25. Madeyski, L., Szala, Ł.: Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study. IET Software Journal 1(5), 180–187 (2007)
26. Marin, M., Moonen, L., van Deursen, A.: An Integrated Crosscutting Concern Migration Strategy and its Application to JHotDraw. In: IEEE International Conference on Source Code Analysis and Manipulation (SCAM 2007), Paris, France (2007)
27. Mens, T., Mens, K., Tourwé, T.: Software Evolution and Aspect-Oriented Software Development, a cross-fertilisation. ERCIM special issue on Automated Software Engineering. Vienna, Austria (2004)
28. Meyer, B.: Object-oriented Software Construction. Prentice-Hall, Englewood Cliffs (1989)
29. Myers, G.J.: Composite/Structured Design. Van Nostrand Reinhold (1978)
30. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)
31. Ponnambalam, K.: Characterization and Selection of Good Object-Oriented Design. In: Workshop on OO Design at OOPSLA 1997, Atlanta, Georgia (1997)
32. Przybyłek, A.: An empirical assessment of the impact of AOP on software modularity. In: 5th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2010), Athens, Greece (2010)
33. Przybyłek, A.: What is wrong with AOP? In: 5th International Conference on Software and Data Technologies (ICSOFT 2010), Athens, Greece (2010)
34. Ribeiro, M., Dósea, M., Bonifácio, R., Neto, A.C., Borba, P., Soares, S.: Analyzing Class and Crosscutting Modularity with Design Structure Matrixes. In: Proceedings of the 21th Brazilian Symposium on Software Engineering (SBES 2007), João Pessoa, Brazil (2007)
35. Sant'Anna, C., Garcia, A., Chavez, C., Lucena, C., von Staa, A.: On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In: 17th Brazilian Symposium on Software Engineering (SEES 2003), Manaus, Brazil (2003)
36. Soares, S., Laureano, E., Borba, P.: Implementing Distribution and Persistence Aspects with Aspect J. In: 17th ACM conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington (2002)
37. Stochmiałek, M.: AOPmetrics, <http://aopmetrics.tigris.org>

38. Störzer, M., Eibauer, U., Schöffmann, S.: Aspect Mining for Aspect Refactoring: An Experience Report. In: Workshop on Towards Evaluation of Aspect Mining at ECOOP 2006, Nantes, France (2006)
39. Störzer, M.: Impact Analysis for AspectJ – A Critical Analysis and Tool-based Approach to AOP. PhD thesis, School of Computer Science and Mathematics, University of Passau, Germany (2007)
40. Sullivan, K., Griswold, W.G., Song, Y., Cai, Y., Shonle, M., Tewari, N., Rajan, H.: Information hiding interfaces for aspect-oriented design. In: 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lisbon, Portugal (2005)
41. Tarr, P., Ossher, H., Harrison, W., Sutton, S.M.: N degrees of separation: multi-dimensional separation of concerns. In: 21st International Conference on Software Engineering (ICSE 2009), Los Angeles, California (1999)
42. Tsang, S.L., Clarke, S., Baniassad, E.L.A.: An evaluation of aspect-oriented programming for java-based real-time systems development. In: 7th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2004), Vienna, Austria (2004)
43. Yourdon, E., Constantine, L.L.: Structured Design: Fundamentals of a Discipline of Computer Program and System Design. Prentice-Hall, Englewood Cliffs (1979)
44. Zhang, C., Jacobsen, H.: Resolving Feature Convolution in Middleware Systems. In: 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, Canada, pp. 188–205 (2004)
45. Zhao, J.: Measuring Coupling in Aspect-Oriented Systems. In: 10th International Software Metrics Symposium, Chicago, IL (2004)

Author Index

- Alves, Tiago 186
Anderson, Ross 1
Arcuri, Andrea 262

Ball, Thomas 141
Balland, Emilie 217
Bendisposto, Jens 50
Boukadoum, Mounir 401
Bracciali, Andrea 96

Cai, Dongxiang 432
Chang, J. Morris 371
Chang, Kai-Hsiang 310
Consel, Charles 217
Crouzen, Pepijn 111
Cunha, Jácome 186

Day, Nancy A. 65
de Halleux, Jonathan 294
Denise, Alain 127
Dietrich, Dominik 81

Ehrig, Hartmut 202
Ermel, Claudia 156, 202
Esmaeilsabzali, Shahram 65

Feng, Lu 2
Fiadeiro, José Luiz 18
Fraser, Gordon 262

Gall, Jürgen 156
Gatti, Stéphanie 217
Gaudel, Marie-Claude 127
Gharaibeh, Bashar 371
Gligoric, Milos 262
Glodt, Christian 171

Hatebur, Denis 232
Heckel, Reiko 341
Heisel, Maritta 232
Hillston, Jane 96
Hoenicke, Jochen 34
Holzer, Andreas 278
Huang, Chung-Hao 310

Januzaj, Visar 278
Jürjens, Jan 232

Kelsen, Pierre 171
Kessentini, Marouane 401
Khan, Tamim Ahmed 341
Kim, Miryung 432
Kugele, Stefan 278
Kwiatkowska, Marta 2

Lambers, Leen 156
Lang, Frédéric 111
Langer, Boris 278
Lassaigne, Richard 127
Latella, Diego 96
Leuschel, Michael 50
Levin, Vladimir 141
Li, Huiqing 356
Li, Juncao 141
Li, Mingshu 416
Lopes, Antónia 18
Lyu, Michael R. 386

Ma, Qin 171
Marinov, Darko 262
Marri, Madhuri R. 294
Massink, Mieke 96

Oudinet, Johan 127

Parker, David 2
Peyronnet, Sylvain 127
Piessens, Frank 247
Podelski, Andreas 34
Post, Amalinda 34
Przybyłek, Adam 447

Rajan, Hridesh 371
Rajan, Sreeranga 326

Sahraoui, Houari 401
Saraiva, João 186
Schallhart, Christian 278
Schmidt, Holger 232
Schröder, Lutz 81
Schulz, Ewaryst 81
Sharma, Rohan 262
Shi, Lin 416

- Taentzer, Gabriele 156, 202
Tautschnig, Michael 278
Thompson, Simon 356
Thummalapenta, Suresh 294
Tillmann, Nikolai 294
Tkachuk, Oksana 326

Vanoverberghe, Dries 247
Veith, Helmut 278
Visser, Joost 186

Wang, Farn 310
Wimmer, Manuel 401
Wu, Jung-Hsuan 310

Xie, Fei 141
Xie, Tao 294, 416

Zhang, Qirun 386
Zheng, Wujie 386
Zhong, Hao 416